Why Should We Care about Data Quality in Software Engineering?

Doctoral Thesis

for the Degree of a
Doctor of Informatics

at the Faculty of Economics,
Business Administration, and
Information Technology
of the
University of Zurich

by
Adrian J. E. Bachmann
from
Emmen, LU, Switzerland

Accepted on the recommendation of
Prof. Abraham Bernstein, Ph.D.
Prof. Dr. Harald C. Gall

2010

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 27, 2010

The Vice Dean of the Academic Program in Informatics:
Prof. Dr. Harald C. Gall

"The only statistics you can trust
are those you falsified yourself."
– *Winston Churchill*

Abstract

Software engineering tools such as bug tracking databases and version control systems store large amounts of data about the history and evolution of software projects. In the last few years, empirical software engineering researchers have paid attention to these data to provide promising research results, for example, to predict the number of future bugs, recommend bugs to fix next, and visualize the evolution of software systems. Unfortunately, such data is not well-prepared for research purposes, which forces researchers to make process assumptions and develop tools and algorithms to extract, prepare, and integrate (i.e., inter-link) these data. This is inexact and may lead to quality issues. In addition, the quality of data stored in software engineering tools is questionable, which may have an additional effect on research results.

In this thesis, therefore, we present a step-by-step procedure to gather, convert, and integrate software engineering process data, introducing an enhanced linking algorithm that results in a better linking ratio and, at the same time, higher data quality compared to previously presented approaches. We then use this technique to generate six open source and two closed source software project datasets. In addition, we introduce a framework of data quality and characteristics measures, which allows an evaluation and comparison of these datasets.

However, evaluating and reporting data quality issues are of no importance if there is no effect on research results, processes, or product quality. Therefore, we show why software engineering researchers should care about data quality issues and, fundamentally, show that such datasets are incomplete and biased; we also show that, even worse, the award-winning bug prediction algorithm BUGCACHE is af-

fected by quality issues like these. The easiest way to fix such data quality issues would be to ensure good data quality at its origin by software engineering practitioners, which requires extra effort on their part. Therefore, we consider why practitioners should care about data quality and show that there are three reasons to do so: (i) process data quality issues have a negative effect on bug fixing activities, (ii) process data quality issues have an influence on product quality, and (iii) current and future laws and regulations such as the Sarbanes-Oxley Act or the Capability Maturity Model Integration (CMMI) as well as operational risk management implicitly require traceability and justification of all changes to information systems (e.g., by change management). In a way, this increases the demand for good data quality in software engineering, including good data quality of the tools used in the process.

Summarizing, we discuss why we should care about data quality in software engineering, showing that (i) we have various data quality issues in software engineering datasets and (ii) these quality issues have an effect on research results as well as missing traceability and justification of program code changes, and so software engineering researchers as well as software engineering practitioners should care about these issues.

Zusammenfassung

In der Softwareentwicklung werden heutzutage diverse Prozess-Hilfsprogramme zur Verwaltung von Softwarefehlern und zur Versionierung von Programmcode eingesetzt. Diese Hilfsprogramme speichern eine grosse Menge an Prozessdaten über die Geschichte und Evolution eines Softwareprojekts. Seit einigen Jahren gewinnen diese Prozessdaten zusehends an Beachtung im Bereich der empirischen Softwareanalyse. Forscher verwenden diese Daten beispielsweise für Vorhersagen der Anzahl Softwarefehler in der Zukunft, für Empfehlungen zur Priorisierung in der Fehlerbehebung oder für Visualisierungen der Evolution eines Software Systems. Unglücklicherweise speichern aktuelle Hilfsprogramme solche Prozessdaten in einer Form, wie sie für Forschungszwecke wenig geeignet ist, weshalb Forscher in der Regel Annahmen über die Softwareentwicklungsprozesse treffen und eigene Tools zum Bezug, Vorbereitung sowie Integration dieser Daten entwickeln müssen. Die getroffenen Annahmen und angewendeten Verfahren zum Bezug dieser Daten sind indes nicht exakt und können Fehler aufweisen. Ebenfalls sind die Prozessdaten in den ursprünglichen Hilfsprogramen von fraglicher Qualität. Dies kann dazu führen, dass Forschungsresultate, welche auf solchen Daten basieren, fehlerhaft sind.

In dieser Doktorarbeit präsentieren wir eine Schritt-für-Schritt Anleitung zum Bezug, Konvertieren und Integrieren von Software Prozessdaten und führen dabei einen verbesserten Algorithmus zur Verknüpfung von gemeldeten Softwarefehlern mit Veränderungen am Programmcode ein. Der verbesserte Algorithmus erzielt dabei eine höhere Verknüpfungsrate und gleichzeitig eine verbesserte Qualität verglichen mit früher publizierten Algorithmen. Wir wenden diese Technik auf sechs Open Source und zwei Closed Source Softwarepro-

jekte an und erzeugen entsprechende Datensets. Zusätzlich führen wir mehrere Metriken zur Analyse der Qualität und Beschaffenheit von Prozessdaten ein. Diese Metriken erlauben eine Auswertung sowie ein Vergleich von Software Prozessdaten über mehrere Projekte hinweg. Selbstverständlich ist die Auswertung wie auch die Publikation des Qualitätslevels, sowie die Beschaffenheit von Prozessdaten uninteressant, sofern kein Einfluss auf Forschungsresultate, Softwareprozesse oder Softwarequalität vorhanden ist. Wir analysieren daher die Frage, wieso Forscher in der empirischen Softwareanalyse sich um solche Gegebenheiten kümmern sollten und zeigen, dass Software Prozessdaten von Qualitätsproblemen betroffen sind (z.B. systematische Fehler in den Daten). Anhand von BUGCACHE, einem prämierten Fehlervorhersage-Algorithmus, zeigen wir, dass diese Qualitätsprobleme einen Einfluss auf Forschungsergebnisse haben können und sich Forscher daher um diese Probleme kümmern sollten. Der einfachste Weg um solche Qualitätsprobleme zu beseitigen wäre die Sicherstellung von guter Datenqualität bei ihrer Entstehung und somit in den Hilfsprogrammen, welche von Beteiligten in der Software Entwicklung (z.B. Software Entwickler, Software Tester, Software Projektleiter, etc.) verwendet werden. Aber wieso sollten diese Personen einen erhöhten Aufwand für eine verbesserte Datenqualität auf sich nehmen? Wir analysieren auch diese Frage und zeigen, dass es drei Argumente dafür gibt: (i) Qualitätsprobleme in Prozessdaten haben einen negativen Einfluss auf die Fehlerbehebung, (ii) Qualitätsprobleme in Prozessdaten haben einen Einfluss auf die Qualität des Softwareprodukts, und (iii) aktuell gültige sowie künftige Gesetze und regulatorische Vorgaben wie beispielsweise der Sarbanes-Oxley Act oder Informatik-Governance Modelle wie Capability Maturity Model Integration (CMMI), aber auch Vorgaben aus dem Management operationeller Risiken, verlangen die Nachvollziehbarkeit sowie Begründung von allen Veränderungen an Informationssystemen. Zumindest indirekt ergeben sich damit auch Anforderungen an eine gute Datenqualität von Prozessdaten, welche die Nachvollziehbarkeit von Änderungen am Programmcode dokumentieren.

Zusammenfassend diskutieren wir in dieser Doktorarbeit wieso

wir uns um Qualitätsprobleme bei Software Prozessdaten kümmern sollten und zeigen, dass (i) Prozessdaten von diversen Qualitätsproblemen betroffen sind und (ii) diese Qualitätsprobleme einen Einfluss auf Forschungsresultate haben aber auch zu einer fehlenden Nachvollziehbarkeit bei Änderungen am Programmcode führen.

Acknowledgements

Writing the acknowledgements is a great pleasure and offers me the opportunity to say thank you to all the people who have supported me on my way finishing this thesis.

First of all, I would like to thank my advisor, Abraham Bernstein, for giving me the opportunity to work on this thesis and for giving me professional support in how to write scientific publications and doing research. Thank you very much for the time I spent as external doctoral student in your group without giving me the feeling of being an outsider. I am absolutely sure that I will miss the time I spent at the institute. I would also like to thank Harald Gall, my co-advisor, for his very valuable feedback and the fruitful discussions.

Many thanks to my collaboration colleagues and co-authors at the University of California in Davis: Premkumar Devanbu, Christian Bird, Foyzur Rahman, Eirik Aune, John Duffy, and Vladimir Filkov. Prem, I learned a lot from you on how to write excellent papers and present disagreeable research results. Thanks also for the funny time we spent together on conferences and visits in Zurich.

A special thank goes to Justin Erenkrantz from the Apache Software Foundation. Justin, you enlightened me with deeply APACHE HTTP WEB SERVER project knowledge in only two days you spent in Zurich and you were a great motivation factor on my way.

This thesis would probably not have been written without prior influences of Peter Vorburger and the support of my employer. Peter, I thank you very much for the opportunity to write an interesting diploma thesis under your advice and for your support during the decision process of whether should I write this thesis as an external doctoral student or not. The time I spent at the institute doing research, writing papers and working on this thesis would not be possi-

ble without the support of the Zurich Cantonal Bank and my (former) managers Felix Rieser, Thomas Frei, and Beatrice Zanella Fux. Thank you very much for giving me the opportunity, needed time and flexibility, as well as the financial support to do my doctor of informatics. In addition, I would like to thank you for the unique opportunity and faith in me to analyze closed source software project data from two real projects of ZKB.

I also would like to thank the DDIS group, particularly Katharina Reinecke, Jonas Tappolet, and Thomas Scharrenbach. Thank you very much for sharing the office with a crazy guy, providing me with very valuable feedback, and being great discussion partners. I will miss the time I spent at IFI sharing an office with you!

Last but not least a big thank you goes to my wife Maria, my parents Trudy and Joe, as well as my sister Karin. Without you and your support I would never have been able to spend the last years with my doctoral studies. Thank you for the understanding and the time you gave me during the holidays, weekends, and weekday evenings I spent doing my research, writing papers, and last but not least, writing this thesis. I love you all!

Adrian Bachmann
Zürich, August 2010

Contents

List of Figures

List of Tables

Part I

Introduction and Related Work

1

Introduction

1.1 Motivation

Software systems today play an important and central role in business as well as private life and are embedded in almost every electronic device. Usually, these software systems are produced by human software engineering experts based on user or business needs and requirements. Unfortunately, humans are imperfect and software systems are usually complex. Hence, software systems may contain bugs and therefore sometimes act not as specified or desired. Most commercial software companies therefore spend much money and effort uncovering all bugs in a software system before it becomes available to the public. But, the increasing software complexity and constraints in time and money do not allow a 100% testing of a software system, software testing rarely uncovers every bug, and "every program has at least one more bug" (Lubarsky's Law of Cybernetic Entomology). Open source software (OSS) projects, in contrast to commercial projects, often pass a testing of software releases by professional testing engineers, but release alpha and beta versions and let users perform testing and reporting/documenting uncovered bugs. The Scan-project of coverity[1], for instance, analyzed the defect density of many popular OSS projects

[1] `http://scan.coverity.com`

and showed that up to 0.5 bugs per 1 000 lines of code (KLOC) are not unusual.

The rising complexity and size of software systems not only constrains a full testing but also has an effect on processes such as the development of software systems, which mostly need the attention of more than one single developer. Hence, efficient handling of bug repair and concurrent development itself increases needs for software tool support. Therefore—as software engineering grew in importance in the last decades—more and more software engineering and testing tools that support practitioners in engineering and maintaining a software system became available. Luckily, these tools (e.g., bug tracking systems, version control systems, integrated development environments) not only support the users and practitioners in their work but also store information about the software engineering and maintenance processes—i.e., bug fixing activities and program changes—and are, therefore, a valuable source of information on the history and evolution of a software system.

Software engineers make use of such data, for example, to define new test cases based on bug reports or to try to understand the history of a software system during bug fixing and refactoring tasks. Since many OSS projects are publicly available and mostly provide free access to the contents of these software engineering tools, software engineering process data have also gained popularity among empirical software engineering researchers during the past few years. These researchers use such data to analyze software engineering and developed approaches and algorithms to evaluate, for instance, the performance of bug fixing processes, predict the number and locale of bugs in future releases, or analyze when a bug was introduced. The location and number of future or hidden bugs, for instance, could be used by project managers to identify the critical parts of a system, limit the gravity of their impact, and facilitate a better planning of testing and engineering efforts (see [Bernstein et al., 2007]) and is, therefore, valuable information.

Due to the rising popularity of empirical software engineering, many workshops, conferences and journals address this topic, for ex-

ample, the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-Evol)[2], International Symposium on Empirical Software Engineering and Measurement (ESEM)[3], Working Conference on Mining Software Repositories (MSR)[4], International Conference on Predictor Models in Software Engineering (PROMISE)[5], and the Empirical Software Engineering journal[6]. Also some of the most important software engineering conferences such as the International Conference on Software Engineering (ICSE)[7], European Software Engineering Conference (ESEC), the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)[8], and Symposium on the Foundations of Software Engineering (FSE)[9] encourage researchers to submit publications in this research area.

In a perfect world, however, all these software engineering tools would be well-integrated in one single software project solution and would allow a tracking from the user requirements, to the changes in the software, its testing, and finally related bugs or changed requirements. Therefore, all program changes, for instance, would be justified by fixes of uncovered bugs or newly developed user requirements. Nowadays, only a few software project solutions such as IBM JAZZ are available and those are, unfortunately, only used by a few software projects. Therefore, in most cases we do not have a perfect world of one single and integrated software project solution but rather an ensemble of different stand-alone systems which provide functionality and support for only a specific part of software engineering and maintenance processes. User requirements are usually documented in text documents or proprietary databases whereas bug tracking systems (BTSs) are usually used to store user requirements and report/-

[2] http://ssel.vub.ac.be/iwpse-evol/
[3] http://esem2010.case.unibz.it/
[4] http://www.msrconf.org/
[5] http://promisedata.org/
[6] http://www.springer.com/computer+science/swe/journal/10664
[7] http://www.icse-conferences.org/
[8] http://www.esec-fse.org/
[9] http://fse18.cse.wustl.edu/

track uncovered bugs. Program code and configuration files are usually managed by version control systems (VCSs), which allow file versioning and concurrent developing.

Whereas fully integrated project solutions such as JAZZ provide integrated data of the project history, stand-alone systems store only their own view on the history of a project and are, in most cases, not integrated. As already mentioned, fully integrated project solutions are not widely used. Therefore, analyzing software engineering process data needs effort to extract, process, and integrate the software engineering process data. In the past years, many researchers presented techniques to do so. Unfortunately, preparing these data is not an exact discipline and requires process assumptions and heuristics by researchers.

Summarizing, much valuable data about the history and evolution of software projects is available in software engineering tools and many researchers make use of such data, providing promising applications and research results. Unfortunately, data gathering and preparation techniques are inexact and the quality of original data itself may be affected by quality issues. In addition, different software projects make use of different kinds of software engineering processes and different kinds of usage of software engineering tools, which may result in varying characteristics of the data across projects. Nonetheless, we believe in an enormous potential of empirical software engineering which is, indeed, supported by research leaders in this field [Godfrey et al., 2009].

1.2 Research Questions and Hypotheses

Software engineering tools such as bug tracking systems (BTSs) and version control systems (VCSs) store large amounts of data on the history and evolution of a software project. Such data has an enormous potential in research and may introduce an important effect on work efficiency and quality assurance of software systems. Therefore, empirical software engineers spend much effort to extract and prepare

BTS and VCS data, provide integrated datasets, and report applications and results. Figure 1.1 shows the work-flow of software engineering data from its origin in software engineering processes to its use in mining software repositories and empirical software engineering applications. In addition, the figure shows how the research ques-

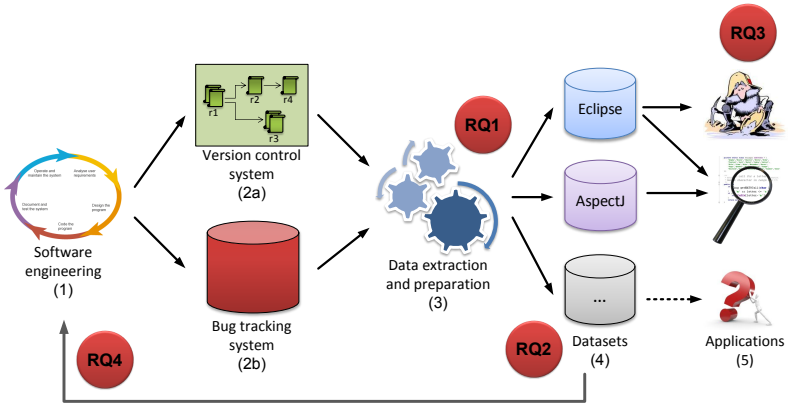


Figure 1.1: Embedding of our research questions among empirical software engineering activities

tions of this thesis are embedded in the whole work-flow. Basically, software engineers develop and maintain software systems following a software engineering process (1) and store process data about the activities in version control systems (2a) and bug tracking systems (2b). Usually, such process data is not well-integrated and sometimes needs to be converted (3), for example, from file-based to transaction-based log files (see discussion in Section 5.3). The integration (i.e., linking) of these data sources is not automatic but has to be done by researchers, typically by scanning through the VCS log messages for potential bug report numbers; conscientious developers enter this information during the check-in process of changed program code (e.g., see [Fischer et al., 2003b]). Unfortunately, such techniques base on heuristics/algorithms and are, therefore, inexact. Checking existent datasets pro-

vided, for example, by Zimmermann *et al.* [Dallmeier and Zimmermann, 2007; Zimmermann et al., 2007], we found that software engineering datasets are plagued by quality issues and that only a fraction of bug reports are mentioned in VCS log files and therefore linked, which leads us to Research Question 1.

RQ 1: *How can we counter the known issues in preparing and linking software engineering process data?*

We analyze and discuss the following hypotheses related to Research Question 1:

HYPOTHESIS 1.1: *Our procedure addresses the known issues in preparing, converting, and linking software engineering process data and enhances existing algorithms.*

HYPOTHESIS 1.2: *Our extended algorithm produces datasets with a higher linking ratio as well as data quality than those previously presented.*

HYPOTHESIS 1.3: *Our data preparation technique produces datasets with a more complete picture of software engineering process data than those previously presented.*

Regarding the datasets (Figure 1.1-4) extracted and prepared as discussed above, unfortunately, little information is available about the quality and the characteristics of them. Furthermore, given the availability of OSS process data, most mining software repositories and empirical software engineering publications report results and algorithms which are developed and tested on OSS project data only (Figure 1.1-5). Consequently, it is unclear how well these approaches and algorithms generalize to closed source software (CSS) projects or even across OSS projects. In particular, differences in the software engineering processes—the matter of testing, developer motivation/in-

centives, and quality assurance—leads us to be careful with assumptions about the applicability of OSS methods for CSS. In summary, this raises the question of how we are able to evaluate the data quality and characteristics of such datasets allowing us to compare the projects.

RQ 2: *How can we qualify and characterize software engineering process data for evaluation and comparison across projects?*

We analyze and discuss the following hypotheses related to Research Question 2:

HYPOTHESIS 2.1: *Our framework of data measures can evaluate and compare the data characteristics and quality across several software projects.*

HYPOTHESIS 2.2: *Software engineering datasets are plagued by data quality issues such as missing information.*

HYPOTHESIS 2.3: *Software engineering datasets vary in their characteristics across projects, especially between open source and closed source software projects.*

However, evaluating software engineering process data quality and characteristics is only the first part of the story. If there is no effect on (i) results of empirical software engineering studies or (ii) the work of practitioners, we can just ignore such issues without any consequences. Since we hypothesize that we have data quality issues in software engineering datasets, we also believe that such data quality issues (e.g., bias in datasets due to incomplete linking) have an effect on research results. With Research Question 3, therefore, we address the question of why researchers should care about data quality in software engineering.

RQ 3: *Why should empirical software engineering researchers care about data quality in software engineering?*

We discuss and analyze the following hypotheses related to Research Question 3:

HYPOTHESIS 3.1: *Process assumptions made by empirical software engineering researchers may be wrong.*

HYPOTHESIS 3.2: *Software engineering datasets are plagued by bias due to a lack of complete linking.*

HYPOTHESIS 3.3: *Bias in software engineering datasets has an effect on empirical software engineering results.*

Poor data quality, indeed, mostly results from the lack of integration or missing information in original software engineering tools. Although researchers are able to deal with data quality issues to some extent, the best way to fix such issues is to ensure accurate data quality at it source. This means, basically, by practitioners who should use tools with better support for data integration or enforcements of rules and regulations in the use of such systems (e.g., not allowing empty commit messages). But why should practitioners change their behavior and spend more money and time to ensure better quality? With Research Question 4 we analyze this question and show why practitioners should change their behavior and ensure better process data quality in the future.

RQ 4: *Why should software engineering practitioners care about data quality in software engineering?*

We analyze and discuss the following hypotheses related to Research Question 4:

HYPOTHESIS 4.1: *Poor software engineering data quality influences the bug fixing process and bug fixing activities (i.e., performance of bug fixing).*

HYPOTHESIS 4.2: *Software engineering process quality (evaluated by process data quality) influences product quality (measured by number of bugs).*

HYPOTHESIS 4.3: *Laws and regulations nowadays require accurate data quality in software engineering (e.g, justification and traceability of all program code changes).*

In addition, promising empirical software engineering research results may not be possible in the future or never become ready for commercial use without having access to datasets with accurate quality.

In summary, we address four research questions and analyze several hypotheses across the work-flow of empirical software engineering. We present an enhanced technique to prepare software engineering process data, evaluate the quality and characteristics of these data, and discuss why empirical software engineering researchers as well as software engineering practitioners should care about data quality issues in software engineering.

1.3 Contributions

Analyzing and discussing our four research questions, we make several contributions affecting both software engineering practitioners as well as empirical software engineers. Our contributions can be summarized as follows:

- Typically, software engineering process data is distributed over several stand-alone systems and, therefore, is not well-prepared for research purposes. The linking of these data requires additional effort by researchers and has to be established by heuristics and algorithms which is, indeed, inexact and critical. In this thesis, therefore, we present a step-by-step procedure on how to extract, convert, and integrate (i.e., link) software engineering process data. We adapted earlier presented techniques resulting

in a higher linking ratio and better data quality at the same time. Nonetheless, we are able to link only a fraction of fixed bug reports, which still may raise threats in applications that use such data.

- Software projects make use of different software engineering processes including varying approaches to test software. Therefore, the data characteristics may vary across different projects—especially between OSS and CSS projects. In addition, empty messages in version control system log files as well as bug reports of low quality (e.g., as reported by Bettenburg *et al.* [Bettenburg et al., 2007b]) raise the question of how we are able to evaluate and compare such datasets. With our data quality and characteristics measures we contribute a framework which enables the possibility to evaluate and compare such datasets. Reporting such evaluation values introduces more transparency in research studies using software engineering process data.

- Poor data quality may affect research results in empirical software engineering, which is why researchers should care. Therefore, we analyze the data quality and characteristics of two CSS and six OSS projects and find, not surprisingly, that all projects are plagued by data quality issues. In addition, we analyze the APACHE HTTP WEB SERVER project in more detail with the assistance of an APACHE expert developer and find that things are even worse: The most relevant bugs in APACHE never show up in the bug tracking system. This may cause serious problems for studies using such data. Also, we introduce two kinds of bias—bug feature bias and commit feature bias—and show that empirical software engineering datasets are affected by both types. We then analyze the impact of such data quality issues on the award-winning BUGCACHE bug prediction algorithm and show that the performance of BUGCACHE is affected by these issues. We therefore conclude that researchers should care about data quality issues in software engineering datasets and should report possible threats in future work using such data.

- Unfortunately, most data quality issues can only be solved by practitioners producing such data. Therefore, we present a set of arguments why software engineering practitioners (e.g., software engineers, project leaders, and test managers) should also care about data quality and ensure better data quality in the future. We discuss three hypotheses: (i) Poor data quality has a negative effect on bug fixing processes, (ii) poor data quality affects the number of bugs (product quality), and (iii) current and future laws and regulations require implicitly accurate data quality in software engineering. In addition, practitioners may profit from future research results provided by researchers such as bug prediction, process analysis, prioritization systems (e.g., which bug should we fix first), etc. Unfortunately, such research and the step in the direction to commercial products and solutions is only possible if we have better data quality for future research.

In summary, as main the contribution we show why empirical software engineers as well as software engineering practitioners should care about data quality in software engineering.

1.4 Thesis Outline

This thesis is structured in six parts:

Part I The remainder of Part I contains a review of the most important related work that is relevant in the context of this thesis. Specifically, we discuss related work on software engineering data extraction and preparation techniques, empirical software engineering applications, work on data quality in software engineering, and a few representative research papers on data quality in other fields. We also include in our discussion on related work recent publications that analyze the interplay of process quality and product quality.

Part II In Part II we briefly discuss a few commonly used software engineering processes as well as often-used tools, introduce the software projects we used in this thesis, and present a step-by-step procedure to extract and prepare (i.e., convert and link) software engineering process data.

In Chapter 3 we briefly discuss commonly used software engineering processes and present what tools are used in these processes, including the data usually stored by such tools.

In Chapter 4 we introduce the six open source and two closed source software projects we used to analyze the research questions and hypotheses in this thesis. We briefly discuss the functionality of the software systems and the tools they use in the process, and present a few software data statistics for each of the projects.

In Chapter 5 we discuss Research Question 1 and show how we extract software engineering process data from these OSS and CSS projects, discuss our extended linking algorithm and show how we convert the file-based VCS log file of CVS into a transacted oriented SVN-like format. In addition, we present the evaluation procedure we used to verify our data extraction and preparation technique.[10]

Part III In Part III we present how we are able to evaluate the data quality and characteristics of software engineering datasets based on a framework of measures. We then use this framework to evaluate and compare the projects presented in Chapter 4.

In Chapter 6 we introduce a framework of twenty data quality and characteristics measures answering Research Question 2. Using this framework, we are able to evaluate the quality and characteristics of pre-existing as well as our own

[10] Major parts of this chapter have already been published [Bachmann and Bernstein, 2009a]

datasets allowing us to test the hypotheses in this thesis.[11]

Chapter 7 makes use of the data quality and characteristics measures framework to evaluate all software projects introduced in the previous part showing that, not surprisingly, all projects are plagued by data quality issues. In addition, we show vast differences in data characteristics across projects, especially between OSS and CSS projects.[12]

Part IV After we evaluated the data quality and characteristics of several projects, in Part IV we focus on Research Question 3 and answer why empirical software engineers should care about these findings.

In Chapter 8 we first go a step deeper and analyze one representing dataset in more detail to find the "ground truth". We engaged an APACHE expert developer who manually annotated a six-week period of the APACHE HTTP WEB SERVER project dataset. Based on the results and interviews, we are able to present a detailed view into the practices of a very popular and often-used OSS project. In addition, we show that software engineering datasets are more plagued by quality issues than thought before and process assumptions made by empirical software engineering researchers may be wrong, such as the most relevant bugs of APACHE never showing up in the APACHE bug tracking system but are instead discussed on the APACHE email discussion system.[13]

Ideally, all bug fixing commits are linked to fixed bug reports and empirical research would consider all type of fixed bug reports. However, even with our extension of the linking algorithm, we are only able to link a fraction of fixed bugs to fixing commits. In Chapter 9 we therefore show that the sample of linked bugs is not representative to all fixed bugs

[11] Major parts of this chapter have already been published [Bachmann and Bernstein, 2009b]

[12] Some of the results have already been published [Bachmann and Bernstein, 2009b]

[13] Parts of this chapter have already been published [Bachmann et al., 2010]

leading to biased datasets. We describe two kinds of bias: Bug feature bias, where only certain types of bugs are linked, and commit feature bias, whereby only certain types bug fixing commits are linked. We then show that our datasets are plagued by both types of bias and that bias has a negative effect on the performance of the award-winning bug prediction algorithm BUGCACHE.[14]

Part V In Part V we address our last research question and discuss why practitioners should care about data quality in software engineering.

In Chapter 10 we show that process data quality affects the performance of bug fixing processes. We analyze if and how data quality and characteristics measures (introduced in Chapter 6) influence each other and may have an effect on bug fixing activities.[15]

In Chapter 11 we discuss how we are able to evaluate process and product quality by empirical methods. In addition, we analyze the hypothesis that process data quality affects product quality. We calculate Kendall Tau rank correlation [Kendall, 1938] values between the data quality measures presented in Chapter 6 and data quality measured by number of bugs, showing weak evidence supporting this hypothesis.[16]

In Chapter 12 we analyze commonly used IT management frameworks, information security standards and guidelines, as well as laws and regulations on their requirements for accurate data quality in software engineering. Among others, we discuss the Sarbanes-Oxley Act [United States Code, 2002], banking laws and regulations (Basel II [Basel Committee on Banking Supervision, 2006]), and the COBIT (Control

[14] Major parts of this chapter have already been published [Bachmann et al., 2010; Bird et al., 2009a]

[15] Parts of this chapter have already been published [Bachmann and Bernstein, 2010]

[16] Parts of this chapter have already been published [Bachmann and Bernstein, 2010]

Objectives for Information and Related Technology) framework [IT Governance Institute, 2007]. We then show that these principles require (implicitly or explicitly) justification and traceability of changes of information systems and therefore changes of (productive) program code. Although such principles are of weak motivation for practitioners to enhance data quality, we strongly believe that future extensions of these principles as well as operational risk management require them to do so, albeit possibly restricted to CSS projects.

Part VI In the last part we discuss the results, limitations, and future work.

Specifically, we summarize and discuss the thesis providing a short overview and contextual discussion in Chapter 13.

In Chapter 14 we discuss limitations leading to possible future work and enhancements.

Chapter 15 concludes this thesis briefly.

The remainder of this thesis contains a glossary, the bibliography, and supplementary information in the appendix.

2

Related Work

The major contribution of this thesis is the evaluation of data quality and characteristics of several datasets often used in empirical software engineering and the effects of data quality issues on software engineering practitioners as well as empirical software engineering researchers. We were motivated by promising empirical software engineering research results and possible threats to them by data quality issues.

This chapter, therefore, briefly reviews the most important related work. We start with a short summary of software engineering data extraction and preparation (i.e., linking) techniques. We adapted and enhanced these techniques in our work and achieved better results. Second, whilst discussing the importance of our research, we explore current research results in the field of empirical software engineering but omit a full discussion due to the enormous number of publications in this field. In Section 2.3, we discuss data quality issues in software engineering. Here we discuss related work on data quality measurement and evaluation as well as field studies in empirical software engineering showing that only a few publications address effects of data quality issues in software engineering. In Section 2.4 we briefly discuss a few publications about data quality issues—such as bias—in other fields. Last but not least, we discuss related work about the interplay of process (data) quality and product quality in software engineering (Section 2.5).

2.1 Software Engineering Data Extraction and Preparation

Software projects usually make use of process tools, for example, to track all changes to the program code and to handle bugs. Process data, for example, bug reports and VCS logs, stored in these systems is very valuable for empirical software engineering research and thus widely used. Unfortunately, software engineering process tools are usually not designed for research purposes. Therefore, the extraction and integration of software engineering process data is critical and has to be done by researchers.

Fischer *et al.* presented a Release History Database (RHDB) which combines VCS data with BTS information and adds missing data not covered by VCSs such as merge points [Fischer et al., 2003b]. They also pointed to the problem of a lack of functionality to support developers with a mechanism for linking bug reports. To link the VCS log and the BTS, Fischer *et al.* searched for VCS log messages which match to a given regular expression[1] and linked them to bug reports. In this first version of the linking algorithm, no further verification of the matching numbers was done. Nonetheless, the integrated view on software engineering process data was novel and is still used in a similar way by many researchers including ourselves.

Later, Fischer *et al.* improved the linking algorithm and built in verification mechanisms to overcome wrong links and, therefore, low quality of the data in the RHDB [Fischer et al., 2003a]. They assumed that wrong links may be created because the context in which a bug report ID is used is not clear and thus bug report IDs might be incorrect or not specified at all. To overcome such incorrect links, Fischer *et al.* developed a link validation method that is based on the bug report ID and the file name affected by a modification. Their validation method rates the confidence of links between VCS log messages and bug reports. An expression such as "bug #42" is rated high because it definitely identifies a bug report. In contrast, a plain six digit num-

[1] For example, "bugi?d?:?=?\s*#?\s*(\d\d\d+)(.*)" or "b=(\d\d\d\d+)(.*)"

ber just appearing in the text of a bug report is rated as low because it could also be something else, such as a date specification. To further improve the correctness of links between VCS log messages and bug reports Fischer *et al.* check the file names specified in VCS log messages and bug reports. Unfortunately, the link-validation used by Fischer *et al.* relies on the confidence by the expressions and the file name information only. In many projects, we do not have any information about changed files or modules and, therefore, are not able to verify the links with this technique.

Čubranić and Murphy developed a tool called Hipikat which integrates information stored in BTSs, VCSs, and email discussion systems [Čubranić and Murphy, 2003; Čubranić et al., 2005]. Hipikat is designed as a tool to help newcomers in an OSS project to become productive faster (see Section 2.2.3 for more details). To integrate bug reports and VCS data, Čubranić and Murphy used a similar approach as Fischer *et al.* did. Specifically, they used a small set of regular expressions to search for expressions commonly used by developers[2] and verified the link candidates by checking whether any activity occurred on the linked bug report within a small time frame (six hours) around the commit, which is quite similar to our approach.

Śliwerski *et al.* adapted the techniques presented by Fischer *et al.* and Čubranić and Murphy [Śliwerski et al., 2005]. They assigned every link two independent levels of confidence: A syntactic level, inferring links from a VCS log message to a bug report, and a semantic level, validating a link via the bug report data. For the syntactic analysis Śliwerski *et al.* used a number of regular expressions[3] to identify potential links to bug reports and set the level of their semantic confidence. This first step is quite similar to previously used approaches. In a second step, Śliwerski *et al.* used a semantic analysis to validate the potential links. Specifically, they used information about the bug

[2]For example, "Fix for bug 1234"

[3]For example, "bug[# \t]*[0-9]+" or "show_bug\.cgi\?id=[0-9]+" and keywords (e.g., "fix(e[ds])", "bugs", "defects")

report [4] and set the level of semantic confidence to each potential link. Based on the levels of confidence, Śliwerski *et al.* decided if a potential link is valid or not.

Schröter *et al.* created an ECLIPSE dataset containing BTS and VCS data and the links between them [Schröter et al., 2006a]. They searched for potential references to bug reports such as "Fixed 42233" or "bug #23444". However, such references have a low trust at first. Therefore, they increased the trust level when a VCS log message contains keywords such as "fixed" of "bug" or matches patterns like "# and a number". The same technique was re-used by Zimmermann *et al.* [Zimmermann et al., 2007]. Compared to previously presented approaches, Schröter *et al.* used a simple technique to validate the links without taking bug report information into account.

German developed a tool called softChange, which is quite similar to Hipikat from Čubranić and Murphy [German, 2004]. He also integrated the data from several sources such as BTSs, VCSs, and email discussion systems. Comparable to previous approaches, German used a single regular expression[5] to identify bug report links in VCS log messages. Although he acknowledges that this approach is error-prone without further verification, no support for such verification was built into softChange.

One of the major drawbacks of CVS (compared to SVN) is that commits (i.e., transactions) are split into individual check-ins. Therefore, CVS does not keep track of which files are committed at the same time (for a full discussion see Section 5.3). German used a sliding window algorithm that recovers the original commits [German, 2004]. He defined two parameters: The maximum length of time for a commit (i.e., transaction), and the maximum distance in time between two file revisions. In an experimental setup, German defined optimum values for both parameters to achieve good results. A similar approach to reconstruct original commits (i.e., transactions) was presented by Zimmermann and Weissgerber [Zimmermann and Weissgerber, 2004] and

[4] For example, the bug report has been resolved as FIXED at least once or the short description of the bug report is contained in the VCS log message

[5] "(\#[0-9][0-9]+|bugs?\s+\#?[0-9][0-9]+)(,\s+\#[0-9][0-9]+)*"

re-used by Breu and Zimmermann [Breu and Zimmermann, 2006]. We also use such an approach but rely only on the time difference between two file revisions as Zimmermann and Weissgerber did.

2.2 Empirical Software Engineering Applications

Software engineering data is a valuable source of information on the history and evolution of a software system. In the past few years, therefore, empirical software engineering became very popular and many interesting and promising results were published (see Section 1.1). We hypothesize that data quality issues may have an impact on such research results. Therefore, we briefly discuss a few papers in this field but omit a full discussion. Specifically, we discuss related work on prediction models, hypothesis testing, and understanding the evolution of a software system. Except for the work by Kim *et al.* [Kim et al., 2007], we have not used these applications and results in our work but they reflect promising research results based on software engineering process data we evaluate in this work and, therefore, may be affected by our findings.

2.2.1 Prediction Models in Software Engineering

Prediction of the Number and Locale of Future Bugs

Bug prediction models are an active topic in empirical software engineering research with a large number of publications at various venues. A recent survey by Catal and Diri [Catal and Diri, 2009] lists almost 100 citations. This topic is also the focus of the PROMISE conference[6] and the Working Conference on Mining Software Repositories (MSR)[7].

[6] See `http://promisedata.org/`
[7] See `http://www.msrconf.org/`

Therefore, many publications exist in this topic and we omit a full discussion, but summarize a few representative publications.

Graves *et al.* attempt to understand the processes by which software ages [Graves et al., 2000]. They used VCS data from a very large, long-lived software system, and explored the extent to which measurements from the change history are successful in predicting the distribution over modules of these incidences of faults. In general, Graves *et al.* showed that process measures based on the change history are more useful in predicting fault rates than product metrics of the code: For instance, the number of times the code has been changed is a better indication of how many faults it will contain than its length is. They also compared the fault rates of code of various ages and found that if a module is, on average, a year older than an otherwise similar module, the older module will have roughly a third fewer faults.

Ostrand *et al.* used a regression model to predict the number of bugs in each file of the release of two large commercial systems [Ostrand et al., 2005]. They gathered data from a database in the form of Modification Requests (MRs). The VCS automatically recorded which files were changed so all MRs were linked to their corresponding program code changes. However, MRs may be related to things other than faults such as enhancement requests or changes in specifications. To overcome this problem, they used heuristics such as number of files changed by an MR to classify it as fault related or not and manually evaluated the results. In the formulation of their study, they initially planned to use MR severity in their prediction model. However, after examining and questioning developers, they abandoned its use due to inaccuracies and inconsistency. Section 3 of [Ostrand et al., 2005] is also an excellent example of identifying and avoiding areas of possible bias in data used for analysis and prediction.

Askari and Holt developed three probabilistic models for predicting future modification of files based on available change histories of software [Askari and Holt, 2006]. In addition, they proposed a rigorous approach for evaluating such predictive models and found that there are differences in the accuracy of the models.

Knab *et al.* presented an approach that applies a decision tree learner on software engineering process data (program code, VCS, and BTS data) for bug density prediction [Knab et al., 2006]. They tried to find underlying rules which can be easily interpreted by humans. To find these rules, they set up a number of experiments to test common hypotheses regarding bugs in software entities. Their experiments showed that a simple tree learner can produce good results with various sets of input data.

Bernstein *et al.* proposed an approach based on a non-linear model on temporal features for predicting the number and location of bugs in program code [Bernstein et al., 2007]. In their experiments, six different models were trained using Weka's J48 decision tree learner (a reimplementation of C4.5). The data they used to evaluate their prediction models were collected from six plug-ins of the ECLIPSE project. These data were then enhanced with temporal information extracted from ECLIPSE's CVS and information from BUGZILLA. Next, a total of 22 features were extracted from these data. These features include items such as the number of revisions and issues reported within the last three months. Using this approach, Bernstein *et al.* successfully showed that the use of a non-linear model in combination with a set of temporal features (which were selected by an automated feature selection algorithm) is able to predict the number and location of bugs with a very high accuracy.

Zimmermann *et al.* analyzed bugs of the ECLIPSE project in detail to answer several questions [Zimmermann et al., 2007]. They showed that the combination of complexity metrics can predict bugs, suggesting that the more complex the code is, the more bugs it has. However, their predictions are far from perfect, which raises the question: Are there better indicators for bugs than complexity metrics?

Kim *et al.* published the famous and award-winning BUGCACHE bug prediction algorithm [Kim et al., 2007]. They analyzed the version history of seven OSS projects to predict the most fault-prone entities and files. The basic assumption was that faults do not occur in isolation, but rather in bursts of several related faults. Therefore, they cached locations that are likely to have faults: Starting from the loca-

tion of a known (fixed) fault, they cached the location itself, any locations changed together with the fault, recently added locations, and recently changed locations. By consulting the cache at the moment a fault is fixed, a developer can detect likely fault-prone locations. This is useful for prioritizing verification and validation resources on the most fault-prone files or entities. In their evaluation of seven OSS projects with more than 200 000 revisions, the cache selects 10% of the program code files; these files account for 73%–95% of faults—a significant advance beyond the state of the art. We re-implement the BUGCACHE bug prediction algorithm to analyze possible effects of data quality issues on prediction results (see Chapter 9).

Prediction of Bug Introducing Activities

Predicting the number and locale of future bugs, as previously discussed, is mainly focused on finding the right place for bug fixing and testing efforts by practitioners. Other researchers try to predict bug introducing activities during software coding. Therefore, fewer bugs should be introduced into the software system that require fixing later.

Zimmermann *et al.* applied data mining to software version histories in order to guide programmers along related changes: "Programmers who changed these functions also changed..." [Zimmermann et al., 2004]. Given a set of existing changes, such rules (a) suggest and predict likely further changes, (b) show up item coupling that is undetectable by program analysis, and (c) prevent errors due to incomplete changes. Specifically, Zimmermann *et al.* developed a prototype tool that suggests further changes to be made and warns about missing changes. The more history data was available to learn by the prototype, the more and better suggestions can be made. After an initial change, the prototype was able to correctly predict 26% of further files to be changed and 15% of the precise functions or variables. The top three suggestions contained a correct location with a likelihood of 64%.

How do design decisions impact the quality of the resulting software? Schröter *et al.* analyzed this question in an empirical study of

52 ECLIPSE plug-ins and found that the software design as well as past failure history can be used to build models which accurately predict failure-prone components in new programs [Schröter et al., 2006b]. Their prediction only requires usage relationships between components which are typically defined in the design phase; thus, designers can easily explore and assess design alternatives in terms of predicted quality. In an ECLIPSE study, 90% of the 5% most failure-prone components, as predicted by their model from design data, turned out to actually produce failures later; a random guess would have predicted only 33%.

Aversano *et al.* used a technique to identify bug introducing changes to train a model that can be used to predict if a new change introduces a bug or not [Aversano et al., 2007]. They represent software changes as elements of an n-dimensional vector space of coordinates extracted from program code snapshots. The evaluation of various learning algorithms on the two OSS projects JHOTDRAW and DNS-JAVA looked very promising, in particular for the K-Nearest Neighbor algorithm, where a significant trade-off between precision and recall was obtained.

In an empirical study, Sahoo *et al.* analyzed the implications of various bug report characteristics on automatic software bug diagnosis tools [Sahoo et al., 2010]. They used several randomly collected reported bugs of six server applications and found that bugs can be reproduced deterministically. In addition, Sahoo *et al.* found that very few input requests are needed to reproduce most of the bugs; in fact, in most cases (78%), just one input request suffices to reproduce the bug. The findings and results discussed in their study can be used in future tools for doing automatic in-production bug diagnosis.

Detection of Security Bugs

Neuhaus *et al.* introduced Vulture, a new approach and tool to predict vulnerable components in large software systems [Neuhaus et al., 2007]. Vulture relates a software project's version archive to its vulnerability database to find those components that had vulnerabilities in

the past. It then analyzes the import structure of software components and uses a support vector machine to learn and predict which imports are most important for a component to be vulnerable. Neuhaus *et al.* evaluated Vulture on the C++ codebase of MOZILLA and found that Vulture correctly identifies about two-thirds of all vulnerable components. This allows developers and project managers to focus their testing and inspection efforts: "We should look at nsXPInstallManager more closely, because it is likely to contain yet unknown vulnerabilities."

In certain software projects bug reporters need to label a bug report as a security bug report or not, to indicate whether the involved bugs are security problems. These security bug reports generally deserve higher priority in bug fixing than non-security bug reports. According to Gegick *et al.* [Gegick et al., 2010], bug reporters often mislabel security bug reports as non-security bug reports partly due to lack of security domain knowledge. This mislabeling could cause serious damage to software-system stakeholders due to the induced delay of identifying and fixing the involved security bugs. To address this important issue, Gegick *et al.* developed a new approach that applies text mining to natural-language descriptions of bug reports to train a statistical model on already manually-labeled bug reports to identify security bug reports that are manually mislabeled as non-security bug reports. Security engineers can use the model to automate the classification of bug reports from large bug databases to reduce the time that they spend on searching for security bug reports. They evaluated the model's predictions on a large Cisco software system with over ten million source lines of code. Among a sample of bug reports that Cisco bug reporters manually labeled as non-security bug reports, their model successfully classified a high percentage (78%) of the security bug reports as verified by Cisco security engineers, and predicted their classification as security bug reports with a probability of at least 0.98.

Prediction of the Bug Severity

The severity of a reported bug is a critical factor in deciding how soon it needs to be fixed. Unfortunately, while clear guidelines exist on how to assign the severity of a bug, it remains an inherent manual process left to the person reporting the bug. Lamkanfi *et al.* investigate whether they can accurately predict the severity of a reported bug by analyzing its textual description using text mining algorithms [Lamkanfi et al., 2010]. Based on three cases drawn from MOZILLA, ECLIPSE, and GNOME, they conclude that given a training set of sufficient size (approximately 500 reports per severity), it is possible to predict the severity with a reasonable accuracy (both precision and recall vary between 0.65–0.75 with MOZILLA and ECLIPSE; 0.70–0.85 in the case of GNOME).

Bug or Enhancement?

Antoniol *et al.* investigated whether the text of the bug reports is enough to classify them into corrective maintenance and other kinds of activities [Antoniol et al., 2008]. For that reason, they used decision trees, naive Bayes classifiers, and logistic regression to accurately distinguish bugs from other kinds of issues. Based on empirical studies performed on issues for MOZILLA, ECLIPSE, and JBOSS they showed that issues can be classified with between 77% and 82% of correct decisions.

Who Should Fix this Bug?

Anvik *et al.* presented a semi-automated approach to assigning bug reports to developers [Anvik et al., 2006]. Their approach applies a machine learning algorithm to the BTS to learn the kinds of bug reports each developer resolves. When a new report arrives, the classifier produced by the machine learning technique suggests a small number of developers suitable for resolving the bug report. With this approach, they have reached precision levels of 57% and 64% on the ECLIPSE and FIREFOX projects respectively.

Lifetime of Bugs

In non-trivial software development projects planning and allocation of resources is an important and difficult task. Estimation of work time to fix a bug is commonly used to support this process but is a difficult task.

Weiss *et al.* presented an approach that automatically predicts the fixing effort, i.e., the person-hours spent on fixing a problem [Weiss et al., 2007]. Their technique leverages existing BTSs: Given a new bug report, they search for similar, earlier reports and use their average time as a prediction. Their approach allows an early effort estimation, helping in assigning bug reports and scheduling stable releases. An evaluation with the JBOSS project showed that the automatic predictions are close to the actual effort.

Panjer used data mining models to predict the time to fix a bug given only the basic information known at the beginning of a bug's lifetime [Panjer, 2007]. For ECLIPSE his models were able to correctly predict up to 34.9% of the bugs into a discretized log scaled lifetime class.

Prediction of Refactoring

Ratzinger *et al.* tried to predict locations of future refactoring based on the development history [Ratzinger et al., 2007]. In an empirical study, they analyzed the OSS projects ARGOUML and SPRING and found that attributes of software evolution data can be used to predict the need for refactoring in the following two months of development. They extracted data from VCSs and BTSs as well as mining features such as growth measures, relationships between classes, and the number of authors working on a particular piece of code. They used this information as input into classification algorithms to create prediction models for future refactoring activities. Ratzinger *et al.* demonstrated that several features such as lines activity rate and number of lines altered per commit provide much information for the assessment of refactorings. But also the structure of the system is crucial for refactorings as the number of co-changed files and the number of files intro-

duced during the maintenance are relevant features. Both ARGOUML and SPRING had these common features although they cover different domains.

2.2.2 Hypothesis Testing in Software Engineering

As discussed in the previous section, software engineering process data is often used for predicting models. In addition, researchers used such data to test specific hypotheses in software engineering. Again, we discuss a few representative publications and remind the reader that all these results rely on empirical software engineering data which may be affected by quality issues as discussed in this thesis.

Koru and Tian analyzed the hypothesis that high-change software modules are more error-prone than modules with only a few changes [Koru and Tian, 2005]. They analyzed the two OSS projects MOZILLA and OPENOFFICE and, contrary to common intuition, found that the top modules in change-count rankings and the modules with the highest measurement values were different. In addition, they observed that high-change modules had fairly high places in measurement rankings, but not the highest places. These findings provided additional guidance in identifying the change-prone modules.

A very interesting study was published by Śliwerski *et al.* [Śliwerski et al., 2005]. They analyzed CVS archives for fix-inducing changes—changes that led to problems, indicated by fixes. In a first investigation of the MOZILLA and ECLIPSE data, Śliwerski *et al.* found that fix-inducing changes show distinct patterns with respect to their size and the day of the week they were applied, with the conclusion to not program on Fridays.

Eaddy *et al.* analyzed the question of whether crosscutting concerns harm code quality (i.e., number of bugs) [Eaddy et al., 2008]. To answer this question, they conducted three extensive case studies and tried to find empirical evidence suggesting that crosscutting concerns cause bugs. They examined the concerns of three small to medium-sized OSS Java projects and found that the more scattered the implementation of a concern is, the more likely it is to have bugs. Eaddy *et al.*

also proposed a theory that suggests why crosscutting concerns might cause bugs, and described their concern model and metrics. Although they found preliminary evidence and all three studies revealed a moderate to strong statistically significant correlation, further studies are needed before we can attempt to draw general conclusions about the relationship between scattering and bugs.

In software engineering it is widely believed that distributed software development is riskier and more challenging than collocated development and therefore more bugs may occur. Bird *et al.* analyzed this conventional belief and examined the overall development of WINDOWS VISTA and comparing the post-release failures of components that were developed in a distributed fashion with those that were developed by collocated teams [Bird et al., 2009b]. They found a negligible difference in failures and concluded that distributed development has little to no effect.

Clones are generally considered bad programming practice in software engineering and identified as a "bad smell" as well as a major contributor to project maintenance difficulties. Rahman *et al.* tried to validate the conventional wisdom empirically to see whether cloning makes program code more bug-prone [Rahman et al., 2010]. Based on BTS and VCS data, they analyzed the relationship between cloning and bug-proneness in APACHE HTTP WEB SERVER, NAUTILUS, EVOLUTION, and GIMP. Rahman *et al.* found that, first, the great majority of bugs are not significantly associated with clones and, second, that clones may be less bug-prone than non-cloned code. Therefore, the study does not support the claim that clones are really a "bad smell".

2.2.3 Understanding Software Evolution

As the third topic in empirical software engineering, researchers try to build tools and visualizing models to better understand the history and evolution of a software project. Such applications may support new developers in more quickly understanding the software, its history, and architecture. In addition, visualization of program code can assist in software review processes and be used to discuss refactoring

tasks in next releases.

Michaud *et al.* developed Shrimp, a tool that integrates and visualizes program code, documentation (Javadoc), and architectural information to aid program code exploration [Michaud et al., 2001]. Unfortunately, Shrimp was build for Java programs only.

Hipikat, which was developed at the University of British Columbia, integrates VCS, BTS and email discussion system data and creates links between these artifacts (see discussion in Section 2.1) [Čubranić and Murphy, 2003; Čubranić et al., 2005]. The integrated information stored in Hipikat provides developers with efficient and effective access to the group memory for a software development project that is implicitly formed by all of the artifacts produced during the development. Based on an ECLIPSE study, Čubranić *et al.* showed that newcomers can use the information presented by Hipikat to achieve results comparable in quality and correctness to those of more experienced members of the team.

German *et al.* developed SoftChange, a tool quite similar to Shrimp and Hipikat, that should aid software engineering research by visualizing data [German, 2004]. SoftChange also integrates data from multiple sources such as VCSs and BTSs (see Section 2.1) and uses visualizations (usually plots) to answer questions such as "How many bugs are closed in each time period?".

Ratzinger *et al.* presented EvoLens [Ratzinger et al., 2005]. EvoLens is a technique to visualize the software as well as metrics of the software over time, which helps developers to understand the evolution of a piece of software. In addition, the visual nature across time facilitates the identification of design erosion and hot-spots of activity, which allows the user to direct perfective maintenance activities to the program code entities involved.

There are many other publications about visualization of the evolution of software systems, for example, to uncover hidden, shifted, or removed dependencies or better understand the software architecture (e.g., [D'Ambros et al., 2005; Fischer and Gall, 2006; Fischer et al., 2003a; Pinzger et al., 2005]).

2.3 Data Quality in Software Engineering

Although software engineering process data is widely used in research (see discussion in previous section), only a few publications cover the quality aspects of these data. In this section, therefore, we briefly review the most relevant related work about data quality in software engineering and address the following three topics: (1) data measurement and evaluation, (2) data quality studies, and (3) dealing with poor data quality.

2.3.1 Data Measurement and Evaluation

While a lot of related literature concentrates on the evaluation and measurement of program code and software quality, only a few publications cover the quality of the software engineering process data such as bug reports and commit logs.

Already in 1978, Cavano and McGall presented a framework for the measurement of software quality [Cavano and McCall, 1978]. They defined software quality factors such as correctness, reliability, efficiency, integrity, usability, maintainability, and testability and provided an in-depth discussion about these factors. Cavano and McGall suggested that such quality factors and quality assurance activities provide early indications of quality problems. Unfortunately, they only addressed software quality in general and left out process (data) quality aspects. Nonetheless, this study provides an excellent overview of software quality aspects.

Basili *et al.* presented the Goal Question Metric (GQM) approach to software modeling and measurement, which emphasizes a purposeful approach to software process improvement, based on goals, hypotheses, and measurement [Basili et al., 1996, 1994]. The approach was originally defined for evaluating bugs for a set of projects in the NASA Goddard Space Flight Center environment. Specifically, the GQM approach consists of three levels: the conceptual level (goal), operational level (question), and quantitative level (metric). A GQM model consists all three levels, starting with a goal (e.g., "purpose:

improvement") which is refined into several questions (e.g., "What is the current change request processing speed?"). Each question is then refined into metrics, some of them objective, some of them subjective. In summary, the GQM approach is a mechanism for defining and interpreting operational and measurable software. The GQM approach is adaptable to different environments and, therefore, has been widely used (see for instance [Grady and Caswell, 1987] and [Nick and Tautz, 1999]) and applied in several organizations, for example, NASA, Hewlett Packard, and Motorola. However, the GQM approach does not define any metrics to evaluate software (process) quality.

In his book [Kan, 2002], Kan provides a good overview of software quality engineering including several quality and project characteristics measures. However, Kan does not seem to provide much information about process data quality and uses mostly simple and well-known characteristics measures such as lines of code (LOC).

Sackmann and Lichter presented a process quality model for the analysis of quality characteristics that is based on evaluating metrics on BUGZILLA, and illustrate it with a comparative evaluation for 25 of the largest products within GNOME [Schackmann and Lichter, 2009]. They suggest that a detailed analysis of the metric results can give valuable advice to the team members on the realistic potential for improvement and also allows the evaluation of the effect of such improvement activities. Unfortunately, their metrics only cover BTS quality and omit VCS as well as linking quality.

2.3.2 Data Quality Studies

In our work we have two major questions: (i) Do we have data quality issues in software engineering datasets, and (ii) such issues have an influence on research results. The first question is covered by a few publications that mainly studied the quality of bug reports. For the second question, in contrast, almost no related work has been published.

Large-scale software products must constantly change in order to adapt to a changing environment. Studies of historic data from legacy

software systems have identified three specific causes of this change: adding new features; correcting faults; and restructuring code to accommodate future changes. In 2000, Mockus and Votta analyzed the hypothesis that a textual description field of a change is essential to understand why that change was performed [Mockus and Votta, 2000]. Also, they expected that difficulty, size, and interval would vary strongly across different types of changes. To test these hypotheses, Mockus and Votta designed a tool which automatically classifies maintenance activity based on a textual description of changes. Developer surveys showed that the automatic classification was in agreement with developer opinions. They found strong relationships between the type and size of a change and the time required to carry it out. Based on the results, Mockus and Votta recommended that a high quality textual abstract should always be provided, especially since we cannot anticipate what questions may be asked in the future. Regarding the effort by researchers to prepare VCS log data for research purposes and the data quality issues we identified in our work, we highly support these early findings.

Chen *et al.* studied the VCS log files of three OSS projects (GNUJSP, GCC-G++, and JIKES) [Chen et al., 2004]. For each VCS log file, they compared the actual changes in the program code to the entries in the VCS log file and discovered significant omissions. The percentage of omissions Chen *et al.* found ranged from 3.7% to 78.6%. The authors suggested that these are significant omissions that should be taken into account when using VCS log files for research. In addition to checking the completeness of the VCS log files, they also checked the correctness of each VCS log entry and found that almost all the entries were correct, although correctness may not be good enough if the data are not complete. We support these quality concerns and show in our work that the VCS log files of other projects have similar quality issues.

Koru and Tian surveyed members of 52 different medium to large-sized OSS projects with regards to bug handling practices [Koru and Tian, 2004]. They found that bug handling processes varied among projects. Some projects are disciplined and require the recording of

all bugs found; others are more lax. Some projects explicitly mark whether a bug is pre-release or post-release. Some record bugs only in program code; others also record bugs in documents. This variation in bug datasets requires a cautious approach to their use in empirical work. We find a similar behavior of bug reporting practices in our APACHE study and, therefore, support these findings.

Ko *et al.* found that many researchers use bug reports as source of information (see discussion above) but none of them have considered how people describe software problems. Therefore, Ko *et al.* analyzed the titles of nearly 200 000 bug reports from five OSS projects (LINUX KERNEL, APACHE, FIREFOX, OPENOFFICE, and ECLIPSE) and discovered several useful trends [Ko et al., 2006]. They found that the titles of the reports generally described a software entity or behavior, its inadequacy, and an execution context, suggesting new designs for more structured report forms. About 95% of noun phrases referred to visible software entities, physical devices, or user actions, suggesting the feasibility of allowing users to select these entities in debuggers and other tools. According to Ko *et al.*, these findings and others have many implications for tool design and software engineering. In our work we did not take the detailed bug description into account but only the status changes and are, therefore, not affected by these issues.

Bettenburg *et al.* provided an extended analysis of bug report quality [Bettenburg et al., 2007a,b]. They investigated the attributes of a good bug report surveying APACHE, ECLIPSE, and MOZILLA developers and used it to develop a computational model of bug report quality. The resulting model was used to build a tool called Cuezilla that allowed the current quality of a bug report to be displayed whilst typing. The survey results suggested that, across all three projects, steps to reproduce and stack traces are most useful in bug reports. The most severe problems encountered by developers are errors in steps to reproduce, incomplete information, and wrong observed behavior. Surprisingly, bug duplicates are encountered often but not considered as harmful by developers. In later work, Bettenburg *et al.* further analyzed the popular wisdom that bug duplicates are a serious problem for OSS projects [Bettenburg et al., 2008]. Specifically, they discussed

several reasons why duplicates may occur in BTSs and showed that the additional information provided by duplicates may help to resolve bugs quicker. According to Bettenburg *et al.*, duplicates should be merged rather than treated. We acknowledge that duplicates may contain additional information if there is an easy way to merge them. Still, we believe that this is a very hard task that needs manual effort by developers. Therefore, we believe that duplicates are, in the first place, risky and time-consuming in bug fixing and should be avoided as much as possible (see Chapter 6).

Hooimeijer and Weimer also analyzed the quality of bug reports and tried to predict whether the bug report will be closed within a given amount of time [Hooimeijer and Weimer, 2007]. Specifically, they presented a descriptive model of bug report quality based on a statistical analysis of surface features of over 27 000 bug reports for the FIREFOX project. The model predicts whether a bug report is triaged within a given amount of time. Interestingly, they found that self-reported bug severity is an important factor in the model's performance but later changes of severity not. This is interesting, because self-reported severity may not be a reliable indicator of a bug's importance.

Schugerl *et al.* discussed the difficulties of writing bug reports of high quality and showed that the quality of bug reports can vary significantly [Schugerl et al., 2008]. In particular, the free form descriptions attached to bug reports often contain important information describing the context of a bug, the type of unexpected behavior that occurs, and even potential solutions to resolve the problem (see discussion above). Therefore, Schugerl *et al.* applied Information Retrieval (IR) and Natural Language Processing (NLP) techniques to measure and predict the quality of the free form descriptions in bug reports. In a case study of ARGOUML, their supervised trained model predicted the quality of bugs reasonably well.

Ayari *et al.* published a MOZILLA case study and attempted to shed some light on threats and difficulties when trying to integrate (i.e., link) BTS and VCS data [Ayari et al., 2007]. They used well-known linking approaches [Fischer et al., 2003b; Śliwerski et al., 2005]

and showed that in MOZILLA only 38% of bugs are actually traced into VCS log messages. We achieved similar results in our case studies and support the finings by Ayari *et al.*

Liebchen *et al.* argued that only a few papers (they identified just one article in the journal of Empirical Software Engineering) address data quality in empirical software engineering and show that data may be problematic for three reasons. First a value may be contaminated by noise, that is the value is inaccurate. Second, a value may be an outlier, that is a highly atypically observation or case. The third reason that data may be considered problematic is if data items are incomplete and values are missing. In [Liebchen et al., 2007], they are concerned with inaccurate values or noise. Liebchen *et al.* investigated the performance of three noise handling techniques in cleaning a large commercial data set. Based on a pilot and a main study they showed that noise is a serious problem and that filtering, robust filtering, and polish improves classification accuracy as well as the quality compared to a "do nothing" approach. Another issue recognized in their study was the impact of missing values.

Paulson *et al.* and Yu *et al.* analyzed differences between OSS and CSS projects [Paulson et al., 2004; Yu and Chen, 2007]. Paulson *et al.* hypothesized that OSS has a higher quality. In addition, they provided five hypotheses, analyzing them with the data of three CSS and OSS projects. They concluded that OSS projects foster more creativity and CSS projects are generally less defective since bugs are found and fixed more rapidly. On the other hand, Yu *et al.* analyzed the average fault (bug) hidden time, average fault pending time, and average fault correction time in CSS projects and OSS projects. They concluded that bugs are fixed more rapidly in OSS projects.

Liebchen and Shepperd surveyed hundreds of empirical software engineering papers to assess how studies manage data quality issues [Liebchen and Shepperd, 2008]. They found only 23 that explicitly referenced data quality. Four of the 23 suggested that data quality might impact analysis, but made no suggestion of how to deal with it. They conclude that there is very little work to assess the quality of data sets and point to the extreme challenge of knowing the "true" values and

populations. They suggest that simulation-based approaches might help.

Effects of poor data quality on empirical software engineering is not widely explored as many studies in this field do not even address the quality topic [Liebchen and Shepperd, 2008]. To our knowledge, only Aranda and Venolia [Aranda and Venolia, 2009] have attempted to verify the completeness and degree of truth in software engineering datasets. Unfortunately, no other published study has addressed this topic and provided an answer to our research questions. Aranda and Venolia provided a field study of coordination activities around bug fixing, based on a survey of software professionals at Microsoft. Specifically, they studied 10 bugs in detail and showed that (i) electronic repositories often hold incomplete or incorrect data, and (ii) the histories of even simple bugs are strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through the automated analysis of software repositories. They report that software repositories show an incomplete picture of the social processes in a project. While they studied 10 bugs in detail, we focus on commit history: We employed an expert to fully annotate a sample of 493 commits of the APACHE HTTP WEB SERVER project.

Summarizing, we can say that a minority of studies explicitly consider the quality in the data. Our work, in contrast, tries to unearth the implications of this behavior by attempting to analyze data quality issues in software engineering datasets.

2.3.3 Dealing with Poor Data Quality

So far we have discussed related work about data quality measurement as well as data quality studies. In this section, we survey a few publications that present ways to deal with data quality issues in software engineering or that try to enhance the quality in the future.

Cartwright *et al.* discussed the problem of missing or highly questionable values in software engineering data [Cartwright et al., 2003]. Naturally this problem is not unique to software engineering, so they explored the application of two existing data imputation techniques

that have been used to good effect elsewhere: Sample Mean Imputation (SMI) and k-Nearest Neighbor (k-NN). In order to assess the potential value of imputation they used two commercial datasets. In both datasets they found that k-Nearest Neighbor (k-NN) and sample mean imputation (SMI) significantly improved the model fit, with k-NN giving the best results. Therefore, Cartwright *et al.* suggested that the k-NN imputation method may have some practical utility for software engineers involved in project effort data collection and analysis.

As discussed in Section 2.3.1, Bettenburg *et al.* [Bettenburg et al., 2007b] presented a computational model of a bug report quality and implemented this model into Cuezilla. They surveyed the quality of bug reports but also presented with Cuezilla a way to achieve better bug reports by assisting bug reporters in achieving a better bug report quality whilst typing. Although Bettenburg *et al.* are not able to deal with existing data quality issues, they presented a comfortable way to ensure better bug report data quality in the future with a beneficial effect for both developers and bug reporters.

Just *et al.* extended their previous work [Bettenburg et al., 2007a,b] and presented the results of a card sort on the 175 comments sent back to them by the responders of the survey [Just et al., 2008]. The card sort revealed several hurdles involved in reporting and resolving bugs, which they present in a collection of recommendations for the design of new BTSs. Such systems could provide contextual assistance, reminders to add information, and most importantly, assistance to collect and report crucial information to developers.

Mockus wrote a whole chapter about how to deal with missing values in software engineering [Mockus, 2008]. He suggested determining the mechanism by which the data are missing and to add observations that may explain why the values are missing. This is important because different conclusions may be reached depending on the particular method chosen to handle missing data. Finally, he discusses deletion, imputation, and multiple imputation techniques to deal with missing data in software engineering.

Not only related to software engineering, Batini *et al.* [Batini et al., 2009] provided a wide range of techniques to assess and improve the

quality of data. Common methodologies in the field of data quality assessment and improvement are presented and systematically and comparatively described.

2.4 Data Quality in Other Fields

Publicly available data sources such as BTSs and VCSs of many software projects have increased the popularity of empirical research in software engineering over the last few years. Unfortunately, as discussed in the previous sections, we have various quality issues in these data that may affect research results. In our work, we mainly focus on bias in software engineering datasets. These issues are not new and bias in data has been considered in other disciplines. Various forms of bias show up, for instance, in sociological and psychological studies of popular and scientific culture. In this chapter, therefore, we discuss very briefly a few publications about bias in other fields.

Confirmation bias where evidence and ideas are used only if they confirm an argument is common in the marketplace of ideas, where informal statements compete for attention [Nickerson, 1998]. Sensationalist bias describes the increased likelihood that news is reported if it meets a threshold of "sensationalism" [Grabe et al., 2001].

Several types of bias are well known: Publication bias, where the non-publication of negative results strengthens incorrectly the conclusions of clinical meta-studies [Easterbrook et al., 1991]; the omnipresent sample selection bias, where chosen samples preferentially include or exclude certain results [Berk, 1983; Heckman, 1979]; and ascertainment bias, where the random sample is not representative of the population mainly due to an incomplete understanding of the problem under study or technology used, and affects large-scale data in biology [Terwilliger and Weiss, 2003].

The bias we study is closest to sample selection bias. Heckmann's Nobel-prize winning work introduced a correction procedure for sample selection bias [Heckman, 1979], which uses the difference between the sample distribution and the true distribution to offset the bias. His

method is only applicable to linear regression models and, while it may apply to some uses of bug data to create bug prediction models, many are not correctable in this manner. MacKinnon and Smith introduced correction methods based on non-linear functions as bias estimators, but they depend on prior knowledge of the bias functions, and even then may increase the variance and overall error [MacKinnon and Smith, 1998]. Of particular interest to our work, and Computer Science in general, is the effect of biased data on automatic classifiers. Zadrozny's studies of classifier performance under sample selection bias show that proper correction is possible only when the bias function is known [Zadrozny, 2004]. Naturally, better understanding of the technologies and methods that produce the data yield better bias corrections when dealing with large data sets, for example, in genomics [Ambroise and McLachlan, 2002].

2.5 Interplay of Process Quality and Product Quality

To answer Research Question 4, we analyze the interplay of process quality and product quality. Specifically, we theorize that such effects may prompt practitioners to ensure better software engineering data quality in the future, allowing more promising empirical software engineering research without having the drawbacks (as discussed above) of poor data quality. In this section, therefore, we review three research papers that explore software engineering process quality and its relation to software quality.

Harter and Slaughter analyzed the question of how quality can be designed into the product [Harter and Slaughter, 2000]. They propose that in manufacturing, process maturity (e.g., consistency and effectiveness of manufacturing processes) is positively associated with product quality. In their work, Diaz and Sligo found initial evidence of a positive relationship between process maturity and software quality at Motorola [Diaz and Sligo, 1997]. Harter and Slaughter designed

their study to address the question of the relationship between process maturity and software quality over the product life cycle. Specifically, they developed a conceptual framework and evaluated the model using archival data collected on CSS products developed over 12 years. The analysis indicates that a higher level of process maturity leads to higher software quality. In summary, their results suggest that software quality is designed into products rather than tested into products. Unfortunately, they analyzed process quality on an abstract level without taking software engineering process data into account on a detailed level as we do.

In recent years, software companies have started to implement software process improvement (SPI) methodologies, of which the ISO 9000 standards [ISO/IEC, 2005b] and the capability maturity model (CMMI) [CMMI Product Team, 2006] are the best known. The underlying principle of both methodologies is to assess organizational capabilities to produce quality software. Whether the practices advocated by these methodologies lead to high-quality software has been the topic of ongoing debates. Ashrafi investigated the impact of such SPI methodologies on software quality, first by theoretical comparison and then with empirical data [Ashrafi, 2003]. With semi-structured interviews, Ashrafi targeted developers who have used CMMI and ISO 9000 and asked them to evaluate the impact of SPI on the quality of the design, performance, and adaption of their software products. Overall, he found that both CMMI and ISO 9000 have a positive effect on software quality. In addition, based on the survey responses, he developed a decision tree that supports the decision process of the best suitable SPI methodology dependent on the software quality goals a company wants to achieve.

Kroeger and Davidson were also motivated by the assumption that the quality of the process will influence the quality, cost, and time-to-release of the software produced. In their paper, they presented a perspective-based model of quality for software engineering processes that was derived from the stated experiences of software engineering practitioners [Kroeger and Davidson, 2009]. Specifically, they interviewed 16 software engineering practitioners in different roles: engi-

neers, engineering managers, engineering academics, process/quality engineers, process/quality managers, and process/quality consultants. Kroeger and Davidson found that software engineering practitioners judge the overall quality of a software engineering process in terms of four distinct quality attributes: suitability, usability, manageability, and evolvability. Unfortunately, the relationship between software engineering process quality and product quality was not taken into account in their study.

2.6 Summary of Related Work

The overview of related work shows that empirical software engineering has an enormous potential with many interesting applications and approaches to enhance software engineering activities in the future. In the first section, we reviewed data extraction and preparation techniques commonly used to prepare software engineering datasets for further research purposes as discussed in Section 2.2. In our work, we use a similar but slightly adapted technique to link BTS with VCS data (see Chapter 5). We then discussed several empirical software engineering applications showing current results in this field. Unfortunately, all these results may be threatened by data quality issues as explored in our work. As discussed in Section 2.3, only a few publications addressed effects of data quality issues and, except for the work by Aranda and Venolia [Aranda and Venolia, 2009], to our knowledge no other study has attempted to verify the completeness and degree of truth in software engineering datasets and explored possible effects on research results. Also, a framework of data quality and characteristics measures that enables the possibility to evaluate and compare software engineering data quality and characteristics across several projects is missing in currently published literature. In Section 2.4 we briefly discussed work on bias in other fields. Work on the relation between process quality and product quality was reviewed in Section 2.5, showing that other researchers found initial evidence for such relations. Unfortunately, we are not able to find evidence in the

data to further support this hypothesis statistically.

Part II

Software Engineering Process Data: Processes, Tools, and Datasets

3

Software Engineering Processes and Tools

To analyze our research questions, software engineering process data is of major importance. Such data accrues in software engineering processes and is stored in tools that automate or support these processes. In this chapter, therefore, we (very) briefly introduce commonly used software engineering processes and discuss major differences between open source software (OSS) and closed source software (CSS) project procedures. As software bug fixing is of major interest, we discuss the activities to fix a bug in more detail followed by a discussion of commonly used tools and systems used in these processes.

3.1 Software Engineering Processes

3.1.1 Software Engineering Processes: A (Very)Short Overview

Nowadays, software projects have the choice of several software engineering processes and development models, such as the Waterfall Development Model, the Prototyping Approach, the Spiral Model, or

the Iterative Development Process Model to list only a few of the most used [Kan, 2002]. Each of these models has its pros and cons and it is up to the development team to adopt the most appropriate one for the project.

We omit a full discussion of all these processes or models and only briefly present the famous (strict) Waterfall Development Model [Royce, 1970] which is often used in large-scale CSS (i.e., commercial) projects. The model consists of seven phases; after each phase is finished, it proceeds to the next one. Transitions to the next phase are often referred to as a "quality gate" that a software project has to pass through. The strict sense of this model makes it inflexible but keeps it simple.

The seven phases are:

1. Requirements specification (Requirements Analysis)
2. Design
3. Implementation (or Coding)
4. Integration
5. Testing (or Validation)
6. Deployment (or Installation)
7. Operation and Maintenance

OSS projects as well as CSS projects in very dynamic application fields mostly prefer other, more flexible development models. Nonetheless, independently of the process or development model, software systems have to be implemented, tested, and maintained. Therefore, the used development model is not that important for our work since we mainly focus on testing and maintenance (bug fixing) activities. In the next sections, therefore, we briefly discuss different testing approaches as well as a commonly used bug fixing process.

3.1.2 Testing Approaches

Software engineering mainly knows two kinds of (pre-release) testing: user testing and professionalized testing. In the old days be-

fore the Internet became popular, fixing (i.e., patching) of software systems after shipping was expensive and laborious. Therefore, companies spent much money and effort on testing and debugging software systems before they were made available to customers, which increased the need for professionalized testing. Indeed, there are still good reasons to test software intensively before shipping, for example, to keep customer satisfaction high. In addition, software systems are embedded in almost every electronic device, which is why professionalized testing can avoid high costs to fix devices (e.g., digital versatile disk (DVD) players, cook steamers, magnetic resonance imaging (MRI) machines, navigation systems) after shipping. On the other hand, software project costs and time-to-market have become very important factors in the software industry and the fixing of software systems with access to the Internet is much easier. Therefore, CSS projects mostly perform professionalized testing but have started to test their products less intensively with the risk of software patches being required later.

OSS (i.e., non-commercial) projects, in contrast, mostly do not have the resources to test software prior to its release but outsource software testing to users. These projects publish new versions as alpha and beta releases and let users perform testing and report bugs. But why should we care about the method of testing?

First, in professionalized testing a few professional testing engineers verify a piece of software based on test cases, test data, test scenarios and report bugs. In user testing, many users may uncover unwanted behavior of the software and may report a bug. In contrast to professionalized testing, the same bug may be reported multiply by different users and duplicates may occur.

Second, professional testing engineers have the knowledge of how a bug report should look like and what kind of information is needed by developers to fix the bug efficiently. In user testing, users may report only a few bugs in their life time and are, therefore, less experienced in how a good bug report should look like.

3.1.3 A Commonly Used Bug Fixing Process

The increasing software complexity and constraints in time and money mostly do not allow full testing of a software system, and software testing rarely uncovers every bug. Therefore, in almost every software system bugs occur and are uncovered by users. Handling and fixing these bugs, therefore, is a critical part of software maintenance and, ideally, each software project should have a well-defined bug fixing process. Crowston defined the following main activities as a commonly used bug fixing process (actors in parentheses) [Crowston, 1997]:

- Find a bug while using system (Customer)
- Attempt to resolve bug (Response Center)
- Attempt to find work-around (Marketing Engineer)
- Diagnose the bug (Software Developer)
- Design a fix for the bug (Software Developer)
- Write the code for the fix (Software Developer)
- Recompile the module and link it with the rest of the system (Integrator)

Usually, this process is performed for every single bug and practitioners may want to track the progress/status of every bug. Therefore, tools that support tracking and managing bug fixing activities are likely to be used.

3.2 Software Engineering Tools

In the previous section, we discussed software engineering processes and development models as well as bug fixing in detail. Usually, these activities are supported by software engineering tools. Modern software project management systems like IBM JAZZ[1] provide the full (or at least partially full) functionality needed to develop and maintain

[1] http://www-01.ibm.com/software/rational/jazz/

a software system in one single system (e.g., bug tracking and program code change management). Additionally, these systems often only allow a change to the program code in combination with a task, which can be a bug that should be fixed (i.e., a bug report), a new feature (e.g., existing feature request), or another task (e.g., refactoring, change copyright in header, etc.). Unfortunately, these mostly commercial systems are not widely used in current software projects and other, mostly stand-alone, systems are used to support the development and maintenance processes of a software system.

Therefore, stand-alone systems such as integrated development environments (IDEs), bug tracking systems/databases (BTSs) and version control systems (VCSs)—which allow concurrent developing and tracking of program code changes—are often used.

Regarding the bug fixing activities discussed in Sub-Section 3.1.3, practitioners use these tools as follows (tools in parenthesis):

- Report a bug (BTS, Figure 3.1-a)
- Dispatch the bug report to a developer (BTS, Figure 3.1-b)
- Check-out the current software version (VCS, Figure 3.1-c)
- Analyze and fix the bug (IDE, Figure 3.1-d)
- Check-in the fixed software version (VCS, Figure 3.1-e)
- Verify the fixed software version against the bug report and change the status of the bug report (BTS, Figure 3.1-d)

For a better understanding of the tools involved, we briefly introduce them in the following sections and also discuss tool-specific details if they are relevant for our work. Since we are interested in software engineering process data, we also discuss what kind of process data these systems store during its use by practitioners.

3.2.1 Integrated Development Environments

Usually, software engineers make use of an integrated development environment (IDE) to develop their software. Modern IDEs provide features such as syntax highlighting, in-program compiling, and debugging to support software engineers, increase work efficiency, and

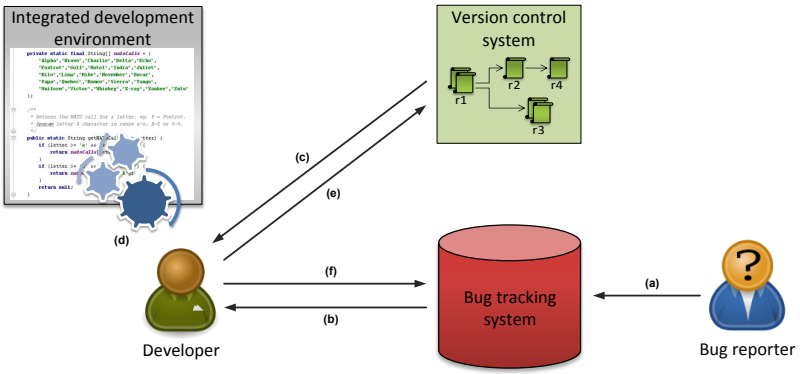


Figure 3.1: Use of software engineering tools (stand-alone systems)

prevent simple bugs. Depending on the programming language, a number of IDEs are available.[2]

However, an IDE is only the work environment for writing/changing the program code. Usually, these systems do not store any information about the development process or the evolution of a project and are, therefore, not important for our data analysis work.

3.2.2 Version Control Systems

As soon as a project exceeds the number of one active developer, a version control system (VCS) is needed to handle all changes to the program code. Several developers may work on the same project concurrently, each one editing files within their own "working copy" of the project, and sending (i.e., committing) their modifications with an optional message to the VCS. If the commit operation succeeds (no conflicts occur), the VCS updates all files involved. In addition, the VCS, writes the user-supplied commit message, the date, and the au-

[2] http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

thor name to its log file. Such systems, like the Concurrent Versions System (CVS)[3] or Subversion (SVN)[4], are widely used in OSS as well as CSS projects. Because VCS log files contain information about all changes to the program code, they are a very valuable source of information about the history and evolution of a software system.

File Based (CVS) vs. Transactional Based (SVN) Versioning

CVS and SVN both provide state of the art functionality but differ from each other significantly in how they versioning the project repository data. CVS firstly addresses the data by location (L) and secondly by time (T), whereas SVN goes the other way:

- CVS: (1) project, (2) location, (3) time → (P:L.T)
- SVN: (1) project, (2) time, (3) location → (P:T.L)

Based on the versioning technique, SVN handles the changes transaction oriented by building a new version/revision of the whole project with every commit (see Figure 3.2), whereas CVS has an independent version/revision on each single file (see Figure 3.3).

Since CVS and SVN are different in the way they store the data, we use the following definition of a commit in the context of this thesis:

DEFINITION: *A commit refers to submitting the latest changes of the program code to the repository.*

Or in other words: A commit accords a transaction.

3.2.3 Bug Tracking Systems

As already discussed, it is almost impossible to release a software system without bugs. Such bugs (or defects) are usually uncovered by users of the software system and reported in a bug tracking system/database (BTS).

[3] `http://www.nongnu.org/cvs/`
[4] `http://subversion.apache.org/`

```
------------------------------------------------------------------------
r653772 | jim | 2008-05-06 15:38:00 +0200 (Tue, 06 Mai 2008) | 2 lines
Changed paths:
   M /httpd/httpd/branches/2.2.x/STATUS
   M /httpd/httpd/branches/2.2.x/modules/ldap/util_ldap.c

PR: 44560

------------------------------------------------------------------------
r653770 | jim | 2008-05-06 15:37:07 +0200 (Tue, 06 Mai 2008) | 2 lines
Changed paths:
   M /httpd/httpd/branches/2.2.x/CHANGES
   M /httpd/httpd/branches/2.2.x/STATUS
   M /httpd/httpd/branches/2.2.x/modules/proxy/mod_proxy_http.c

PR 44165

------------------------------------------------------------------------
```

Figure 3.2: SVN log file (verbose; non-XML; example of the APACHE HTTP WEB SERVER project)

```
RCS file: /cvsroot/eclipse/org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/memory/renderings/HexRendering.java,v
Working file: org.eclipse.debug.ui/ui/org/eclipse/debug/internal/ui/views/memory/renderings/HexRendering.java
head: 1.11
[...]
revision 1.10
date: 2007-01-20 00:10:46 +0100;  author: schan;  state: Exp;  lines: +2 -83;  commitid: ccb45b14ff64567;
Bug 114377:  [Memory View] Endian in hex view and ASCII view doesn't work
----------------------------
[...]
----------------------------
revision 1.1
date: 2005-02-09 18:32:56 +0100;  author: darin;  state: Exp;
Bug 84799 - Implement Memory View and renderings with new rendering APIs
=============================================================================
```

Figure 3.3: CVS log file (example of the ECLIPSE project)

Modern BTSs support the whole bug fixing process (as a part of the software maintenance processes) by providing a bug report status model, possibilities to discuss an issue, and track the fixing progress. Non-commercial ("free") open source BTSs such as BUGZILLA[5] or ISSUEZILLA[6] are very popular in OSS projects, whereas in CSS projects commercial bug tracking and testing suites such as HP QUALITY CENTER[7] or Atlassian JIRA[8] are more likely used.

[5] http://www.bugzilla.org/

[6] IssueZilla is no longer available for download.

[7] https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1131_4000_100__

[8] http://www.atlassian.com/software/jira/

All these BTSs provide the functionality of a modern BTS and support, for instance, a bug report status model (bug report life cycle) as shown in Figure 3.4 for BUGZILLA. In addition, all BTSs investigated, store a set of common attributes for each of the bug reports:

- Bug ID (unique identification number)
- Summary and description of the bug
- Reporter name and/or email address
- Assignee name and/or email address
- Current status of the bug (e.g., new, verified, assigned, closed, etc.)
- Resolution of the bug report (e.g., fixed, duplicate, works-for-me, etc.)
- Priority of the bug
- Date of reporting the bug
- Further product-specific attributes such as operating system, hardware, web browser, etc.
- Attachments such as screenshots, stack-traces, etc.
- Bug report related comments (discussion)

Depending on the BTS, additional attributes such as severity or bug type (e.g., bug, feature-request, etc.) are supported. BTSs do not only store the current status of a bug report, changes to the bug report are also tracked and stored in a so called activity log file.

In summary, these systems store detailed information about bug fixing activities in a software project and are, therefore, a valuable source of information.

Since testing is outsourced to users in many OSS projects, everyone needs to have access to the BTS and should be able to report bugs. Most OSS projects, therefore, have a BTS available on the Internet and only in some cases is a registration needed, but there are typically no limitations on who is allowed to report a bug. Only security relevant bug reports are, in some projects, limited in their access to certain users.

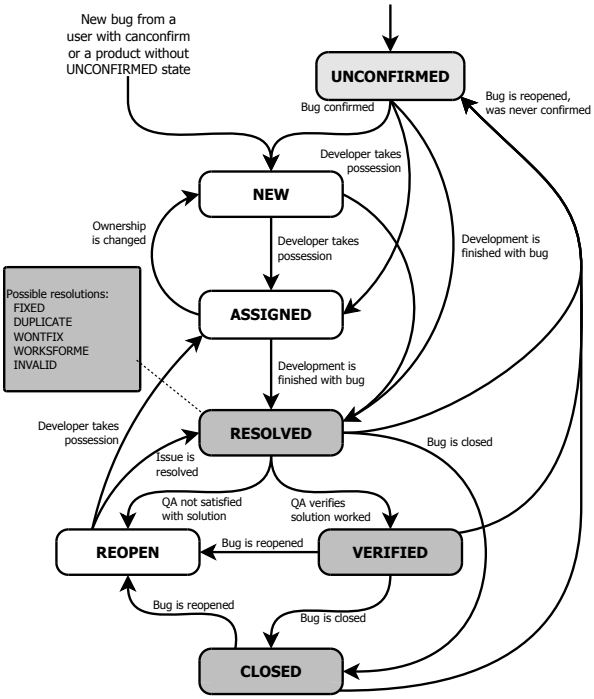


Figure 3.4: BUGZILLA: Life cycle of a bug [Mozilla Foundation, Bugzilla, 2010]

3.3 Concluding Discussion

In this chapter we discussed often-used software engineering and bug fixing processes. We also showed that these processes are usually supported by software engineering tools which store valuable data during the process. Therefore, software engineering processes are usually well-documented by data and these systems are valuable sources of information about the evolution and history of a software project. The combination of these information sources (e.g., BTSs and VCSs) pro-

vides even more valuable information, for example, what changes to the program code were performed to fix a specific bug. Unfortunately, the integration of these systems is not automatic but has to be maintained manually by the developers (see Figure 3.1). Conscientious developers, for instance, refer to a given bug report in the BTS by typing the bug number in the commit message of the VCS.

Software project management systems like JAZZ provide the full (or at least partial) functionality needed to develop and maintain a software product. Therefore, the process data generated by these systems is well-integrated, assuming that these systems are used properly (e.g., no changes linked to empty work items). Unfortunately, these systems are not widely used in current software projects and therefore little data from these systems is available.

In the next chapter we discuss which software projects we used to analyze our research questions and, in Chapter 5, we present a step-by-step procedure to extract and prepare the data from the tools introduced in this chapter.

4

Investigated Software Project Datasets

Analyzing our research questions and hypotheses needs software engineering process data. In the previous chapter we introduced the tools that store such data. In this chapter we now present the software projects we used in our work. We selected six popular and often-used OSS as well as two medium- to large-scale CSS projects:

- APACHE HTTP WEB SERVER (OSS)
- ECLIPSE IDE (OSS)
- GNOME Desktop Suite (OSS)
- NETBEANS IDE (OSS)
- OPENOFFICE (OSS)
- MOZILLA (OSS)
- BSZKB#1 – Banking System 1 (CSS)
- BSZKB#2 – Banking System 2 (CSS)

All these projects have a long project history with many users or systems involved and play an important role in their field. The selected OSS projects are widely used and well known in empirical software engineering research. Many studies in this area rely on them. In

addition, we selected two CSS datasets provided by the Zurich Cantonal Bank[1]. These two CSS datasets allow a slight insight into the commercial practices/processes, the resulting data quality, and characteristics and comparisons to OSS datasets.

The considered OSS projects make use of BUGZILLA or ISSUEZILLA as BTS and SVN or CVS as VCS. For all OSS projects, both systems are available on the Internet and allow free and open access to their contents. The CSS projects, on the other hand, make use of QUALITY CENTER or JIRA as BTS and SVN as VCS. In contrast to the OSS projects, only a few people have the permission to access and modify the VCS and BTS data. Therefore, this data is not available to the public.

Specifically, we used the procedure discussed in the next chapter to extract and prepare the data and created a process dataset for every software project introduced above. Table 4.1 lists some basic statistics for each of the project datasets and shows the time periods we considered.

In the next sections, we briefly discuss every software project in more detail. In addition to our own datasets, we also used pre-existing datasets provided by other researchers for comparison and validation reasons. We introduce these datasets in Section 4.3.

[1] http://www.zkb.ch

Table 4.1: Details of software projects investigated ("#" = number of)

	APACHE	ECLIPSE	GNOME	NETBEANS
OSS / CSS	OSS	OSS	OSS	OSS
BTS	BUGZILLA	BUGZILLA	BUGZILLA	ISSUEZILLA
VCS	SVN	CVS	SVN	CVS
Considered time period	2002-03-18 to 2008-04-28	2001-10-11 to 2008-02-29	2000-05-18 to 2008-09-30	2000-06-05 to 2008-04-30
#Weeks	319	333	436	412
#Bug reports	4 997	215 298	492 107	127 421
#Fixed bug reports	1 439	112 309	113 303	66 786
#Duplicate bug reports	619	28 052	144 020	18 890
#Bug report activities	19 152	1 412 467	1 973 620	923 764
#Bug report comments	17 900	929 056	1 266 172	568 788
#Bug report attachments	1 586	89 250	N/A	60 317
#Different bug reporters	3 510	18 836	158 561	11 410
#Commit messages (transactions)	16 546	221 156	655 668	378 284
#Different developers (committers)	75	187	1 503	648

	OPENOFFICE	MOZILLA	BSZKB#1	BSZKB#2
OSS / CSS	OSS	OSS	CSS	CSS
BTS	ISSUEZILLA	BUGZILLA	QUALITY CENTER	QUALITY CENTER
VCS	CVS	CVS	SVN	SVN
Considered time period	2000-10-21 to 2008-04-30	1998-10-01 to 2009-08-31	2005-03-18 to 2008-02-29	2007-04-03 to 2009-07-30
#Weeks	392	569	154	123
#Bug reports	88 837	495 985	7 843	640
#Fixed bug reports	34 586	158 386	4 449	304
#Duplicate bug reports	14 319	129 069	108	13
#Bug report activities	684 988	5 384 816	114 109	8 108
#Bug report comments	579 747	3 672 984	18 315	1 454
#Bug report attachments	53 219	389 868	8 606	208
#Different bug reporters	19 707	122 325	133	39
#Commit messages (transactions)	106 710	220 460	24 045	22 652
#Different developers (committers)	122	651	51	49

4.1 Used Open Source Software Project Datasets

4.1.1 APACHE HTTP WEB SERVER

The APACHE HTTP WEB SERVER project[2] develops and maintains an OSS HTTP server for modern operating systems. The project aims to provide a secure, efficient, and extensible server that provides HTTP services conformant to current HTTP standards. APACHE is developed and maintained by an open community of developers under the auspices of the Apache Software Foundation and is characterized as OSS. APACHE has been the most popular web server on the Internet since April 1996. As of July 2010, APACHE served over 54.79% of all websites and over 66.60% of the million busiest [Netcraft Ltd., 2010].

Based on its popularity as a web server, APACHE is one of the most popular OSS projects and widely used in current empirical software engineering research (e.g., [Bettenburg et al., 2007b; Just et al., 2008; Ko et al., 2006; Mockus et al., 2002; Paulson et al., 2004; Yu and Chen, 2007]). Therefore, we believe that this project is representative of many OSS projects used in empirical software engineering, and thus a good subject for an in-depth examination of data quality.

Evaluation Sample Dataset

To test some of the hypotheses and truly understand bug reporting and committing practices, we must uncover the ground truth: We must analyze completely (at least a time-window of) the commit version history of a project, and precisely identify all the commits that are bug fixes, and those that are not.

To get at this ground truth requires skill, knowledge, and effort: One must compare successive versions, understand the change, identify any relevant reported bugs in the VCS, and manually establish a link when possible. This process must be repeated until we have a

[2] http://httpd.apache.org/

large enough sample for statistical analysis. Unfortunately, this currently requires an expert to manually extract the information from multiple sources and analyze it.

With our own (rather modest) resources, we could only completely evaluate and manually verify a subset of the original APACHE dataset. Therefore, we had to sample the original dataset. There were two choices: random sampling or temporal sampling.

Random sampling requires some rationale for selecting a sample, for example, prior knowledge of the distribution of the relevant co-variates to the study, so that a sample representative of the population could be chosen. It is difficult to decide a priori what such co-variates might be, let alone their distribution. So, we chose to perform *temporal sampling*.

With this approach, we chose to verify all the commits in a given period. With complete results for that period, we can then revisit our earlier results and judge the quality against this limited but complete and accurate temporal sample.

To find a "typical" period for our evaluation sample dataset we analyzed the whole original APACHE dataset based on week-long periods. Then, we chose a period of six consecutive weeks that was as representative as possible to the overall original APACHE dataset in terms of its descriptive process statistics (e.g., similar proportions of bugs and commits). Table 4.2 lists some basic software process statistics for both the original and the evaluation sample APACHE datasets including the finally defined time-frames.

Later in this work we show how we used and verified this APACHE evaluation sample dataset. We engaged the services of an APACHE expert developer, Dr. Justin Erenkrantz, as an informant to verify our APACHE dataset and manually annotate all commits of the evaluation sample dataset (using a tool called LINKSTER [Bachmann et al., 2010]). Clearly, the quality of this completely annotated evaluation dataset is predicated on the expertise of the annotator. Our informant, Justin, is a expert developer of the APACHE HTTP WEB SERVER project (since January 2001), the President of the Apache Foundation, and serves on the Foundation's Board of Directors. He also develops for Apache

Portable Runtime, Apache flood and Subversion[3]. Based on Justin's familiarity with the APACHE project, we have high confidence that the results of our evaluation sample dataset are trustworthy. The results and findings of our evaluation and verification work we discuss in the next chapters.

Table 4.2: Details to the APACHE datasets ("#" = number of)

Dataset	Original	Evaluation Sample
Considered time period	2002-03-18 to 2008-04-30	2005-09-23 to 2005-11-18
#Bug reports	4 997 (100%)	103 (100%)
#Fixed bug reports	1 439 (28.80%)	23 (22.33%)
#Linked bug reports	686 (13.73%)	10 (9.71%)
#Duplicate bug reports	619 (12.39%)	8 (7.77%)
#Invalid bug reports	1 730 (34.62%)	38 (36.89%)
#Different bug reporters	3 510	98
#Commit messages (transactions)	16 546 (100%)	493 (100%)
#Empty commit messages	4 (0.02%)	0 (0.00%)
#Linked commit messages	1 034 (6.25%)	29 (5.88%)
#Different developers (committers)	75	23

4.1.2 ECLIPSE IDE

The ECLIPSE project[4] was originally created by IBM in November 2001 as OSS and supported by a consortium of software vendors. In January 2004, the Eclipse Foundation was created as an independent non-profit corporation to act as the steward of the Eclipse community.

[3] See `http://www.erenkrantz.com/` for more details.
[4] `http://www.eclipse.org/`

Nowadays, ECLIPSE is a very popular multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others. Nowadays, ECLIPSE is still under the auspices of IBM.

ECLIPSE data is well known and used in many empirical software engineering studies (e.g., [Anvik et al., 2006; Bettenburg et al., 2007a,b; Breu et al., 2006; Joshi et al., 2007; Just et al., 2008; Ko et al., 2006; Moser et al., 2008; Panjer, 2007; Čubranić et al., 2005; Zimmermann et al., 2007]) as well as in the Mining Software Repositories Conferences' mining challenges in the years 2007 and 2008 (see Section 4.3).

4.1.3 GNOME Desktop

The GNOME Desktop project[5] offers an easy to understand desktop environment for Linux or UNIX computers and is composed entirely of free and open source software. The GNOME project puts heavy emphasis on simplicity, usability, and making things "just work". The other aims of the project are: freedom, accessibility, internationalization and localization, developer-friendliness, organization, and support (refer to the GNOME web site for more details).

In empirical software engineering, GNOME is not as popular as ECLIPSE but nonetheless likely used in several publications (e.g., [Linstead and Baldi, 2009; Schackmann and Lichter, 2009; Yu and Chen, 2007]) as well as in the Mining Software Repositories Conferences' mining challenges in the years 2009 and 2010. Specifically, we decided to use GNOME due to similar functionality to two of our CSS projects.

[5] http://www.gnome.org/

4.1.4 NETBEANS IDE

The history of NETBEANS[6] began in 1996 as Xelfi, a Java integrated development environment (IDE) student project under the guidance of the Faculty of Mathematics and Physics at the Charles University in Prague. In 1997, Roman Staněk formed a company around the project and produced commercial versions of the NetBeans IDE until it was bought by Sun Microsystems in 1999. Sun open-sourced the NetBeans IDE in June of the following year. The NetBeans community has since continued to grow, thanks to individuals and companies using and contributing to the project. Nowadays, NETBEANS is a powerful IDE for developing with Java, JavaScript, PHP, Python, Ruby, Groovy, C, C++, Scala, Clojure and many more (for a complete overview refer to the web site of NETBEANS) and is comparable to ECLIPSE.

In research, the NETBEANS project is seldom used and only a few publications using NETBEANS are available (e.g., [Ekanayake et al., 2009]). Nonetheless, we decided to use NETBEANS for two reasons: First, NETBEANS and ECLIPSE have similar purposes, which enables interesting comparisons of data characteristics and quality between these two IDEs. Second, NETBEANS was originally developed as a CSS project and still has a slight touch of a CSS project. Again, the evaluation of data for this project seems to be surprising.

4.1.5 OPENOFFICE

OPENOFFICE[7] is an office suite originally developed under the brand StarOffice by StarDivision. In August 1999 Sun Microsystems acquired the suite and released the program code in July 2000 with the aim of reducing the dominant market share of Microsoft Office by providing a free and open alternative. The OPENOFFICE project is primarily sponsored by the Oracle Corporation (which acquired Sun Microsystems) and other major corporate contributors such as Novell, RedHat, RedFlag CH2000, IBM, Google, and others. Nowadays, OPENOFFICE is

[6] http://netbeans.org/
[7] http://www.openoffice.org/

available for a number of different computer operating systems and supports the ISO/IEC standard Open Document Format (ODF) for data interchange as its default file format, as well as Microsoft Office formats among others. As of November 2009, OPENOFFICE supports over 110 languages.

In empirical software engineering, the office suite is used in several publications (e.g., [Bakota et al., 2006; Canfora and Cerulo, 2005; Ekanayake et al., 2009; Ko et al., 2006; Koru and Tian, 2005]) and is similar to one of our CSS projects investigated in this thesis.

4.1.6 MOZILLA Project

MOZILLA[8] is an OSS project that creates and maintains many popular and innovative products such as the Firefox web browser, the Thunderbird email application, and the bug tracking tool BUGZILLA. The MOZILLA Organization was founded by Netscape Communications Corporation in 1998, before their acquisition by AOL, with the goal of creating a new Internet suite based on the program code of Netscape Communicator. On July 15, 2003, the organization was formally registered as a non-profit organization, and became Mozilla Foundation.

Software engineering process data from the MOZILLA project is often used in research (e.g., [Antoniol et al., 2004; Bettenburg et al., 2007b; Gyimothy et al., 2005; Hooimeijer and Weimer, 2007; Just et al., 2008; Knab et al., 2006; Mockus et al., 2002]) since the project offers several different applications and develops the famous BUGZILLA bug tracking solution (see discussion in Section 3.2.3). Compared to the other projects, we decided to use the full MOZILLA project including all applications for our work to have one additional dataset of the size of GNOME.

[8] http://www.mozilla.org/

4.2 Used Closed Source Software Project Datasets

In contrast to OSS projects, which usually allow free and open access to their systems, CSS projects usually only grant a few people the permission to access and modify their software engineering process data. Therefore, this data is not available to the public and researchers have to negotiate terms and conditions with companies to gain access to such data, which is usually a time-consuming task. We were able to convince the management of a large Swiss bank for a partnership and, therefore, received the permission to use its data for our research purposes. Unfortunately, due to security and confidentiality concerns, we are not allowed to publish detailed information to the CSS datasets. Two CSS projects provided by the Zurich Cantonal Bank are medium-scale banking software systems with many users or systems involved and have various releases over many years.

4.3 Pre-Existing Datasets

For some of our experiments as well as for comparison and validation reasons, we used our own ECLIPSE as well as the pre-existing ECLIPSE_Z bug dataset from the University of Saarland [Dallmeier and Zimmermann, 2007; Zimmermann et al., 2007]. The ECLIPSE_Z dataset[9] is well-documented and has been widely used in research (e.g., [Moser et al., 2008; Čubranić et al., 2005; Zimmermann et al., 2007]) as well as in the Mining Software Repositories Conferences' mining challenges in the years 2007[10] and 2008[11]. The pre-existing ECLIPSE_Z dataset is also available for download on the Internet.[12]

[9] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse (release 1.1)
[10] http://msr.uwaterloo.ca/msr2007/challenge/
[11] http://msr.uwaterloo.ca/msr2008/challenge/
[12] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

4.4 Concluding Discussion

In this chapter we presented the six OSS as well as two CSS datasets we use to analyze our research questions and hypotheses. The OSS projects are well known and often used in empirical software engineering. In addition, we presented two CSS projects, which allow a comparison between OSS and CSS practices/processes. We also introduced our evaluation sample dataset for the APACHE project. We will use this dataset to evaluate (i) our data extraction and preparation technique presented in the next chapter and (ii) commit feature bias in software engineering datasets (Chapter 9).

5

Data Extraction and Preparation[1]

In Chapter 3 we discussed the software engineering tool used by the software projects investigated in this thesis. In addition, we introduced the problem of the missing integration of these tools as well as the circumstance that SVN and CVS use different techniques for versioning data in the repository. Since these software engineering tools are specified and implemented to support software engineering practitioners, the extraction of data from these systems for empirical software engineering is not well-supported but has to be done manually by researchers.

In summary, in using such software engineering process data for empirical software engineering research, researchers have to counter the following issues:

1. missing export or data handling functionality (e.g., API, extended export, and reporting functionality)
2. non-transaction oriented data (in the case of CVS), and
3. lack of data integration (i.e., linking of VCS data with BTS data).

[1]Major parts of this chapter have already been published [Bachmann and Bernstein, 2009a]

In this chapter we therefore introduce a step-by-step procedure to extract and prepare software engineering process data and counter these issues. We extend previously presented approaches (e.g., [Fischer et al., 2003b; Śliwerski et al., 2005; Čubranić and Murphy, 2003; Zimmermann et al., 2007]), answer Research Question 1, and analyze the following hypotheses:

HYPOTHESIS 1.1: *Our procedure addresses the known issues in preparing, converting, and linking software engineering process data and enhances existing algorithms.*

HYPOTHESIS 1.2: *Our extended algorithm produces datasets with a higher linking ratio as well as data quality than those previously presented.*

HYPOTHESIS 1.3: *Our data preparation technique produces datasets with a more complete picture of software engineering process data than those previously presented.*

Based on an evaluation of our approach (Section 5.5), we are able to support all hypotheses and answer Research Question 1.

5.1 Data Retrieval

First, we have to extract the data from the software engineering tools. As discussed in Section 3.2.2, a VCS usually stores a log file that contains the following information for each commit/committed file:

- author,
- date and time,
- changed files,
- and an optional log message.

Since we have usually at least read-only access to the VCS, the procedure to extract the log file is straight forward: Commonly used VCSs such as CVS or SVN allow to retrieve this log file by using one single command. Whereas CVS only supports simple text files as output format, SVN also allows the log file to be retrieved in an XML file format, which simplifies the parsing of the data. Note that there are vast differences between CVS and SVN in how these systems version the data and store the log file (see Section 3.2.2). We discuss the handling of these differences in the next section.

The procedure to retrieve the process data from BTSs needs more attention. If we have access to the BTS underlying database system—which is mostly the case in CSS projects—we can perform a database dump operation to retrieve all relevant data. Unfortunately, in most cases we are not allowed to access the underlying database system unless we have an agreement with the operator of the BTS (which is usually only the case in CSS projects). Luckily, in OSS projects, the BTSs are mostly available on the Internet and readable to the public—often even without registration (see Section 3.2.3)—and provide all the bug information needed in the HTML as well as partially in an XML format. Therefore, we can retrieve the bug report data including all bug report changes over the web interface by fetching and parsing all provided data files (XML and/or HTML). Specifically, we use a wget[2] based shell-script to download all the relevant files from the BTS.

Downloading these data, we have to ensure we receive consistent data from the BTS and the VCS. Often there are many different projects in one single BTS instance (e.g., in ECLIPSE). Therefore, we have to ensure only the relevant data is obtained from each data source. In addition, some of the bug reports (e.g., security relevant bug reports) are not open to the public and usually cannot be retrieved from the BTS.

[2] `http://www.gnu.org/software/wget/`

5.2 Data Parsing

To access the data for our needs, we have to parse and store the data retrieved—as discussed in the previous section—in a practical way. Therefore, we decided to store all the process data in a relational database system which allows us to extract, combine, and select the data in every desired format.

To parse the VCS data, we developed a CVS and SVN log file parser that stores all the relevant information from the log files in a relational database system. As already mentioned in the previous section, SVN provides the log file optionally in an XML format, which can be parsed quite easily with an XML parser/parser library.[3] Unfortunately, CVS does not provide such functionality but supports instead a simple text file format. Therefore, our VCS log file parser includes a simple text parser which extracts all the relevant attributes for each revision out of the CVS log file.

To extract BTS information, we use a similar approach if we had no access to the BTS database system. Again, we developed a parser which runs through all the downloaded files and uses the SAX XML parser to extract the relevant data from the XML files. To extract the relevant information from the HTML files we use a simple text parsing technique. Luckily, the relevant information is stored in a HTML table, which makes the text parsing algorithm simple and less error-prone. If we have access to the BTS underlying database system and are able to perform a database dump operation, we can load this database dump directly back into our relational database without any additional parsing effort.

5.3 Data Conversion

To analyze our research questions and hypotheses, we are interested in the data that reflects the software engineering processes. Therefore,

[3] For example, `http://www.saxproject.org/`

we need to obtain a process-oriented (or in other words transaction-oriented) view of the data that maintains its temporal flow.

DEFINITION: *A transaction accords the activity of sending information to a VCS by someone in one single work step (i.e., commit).*

Luckily, the BTSs discussed in Section 3.2.3 provide the data already in this format. Analyzing the VCS log files can be a bit more involved since CVS and SVN make use of different approaches to versioning the data (see Section 3.2.2) and have, therefore, different log files. Whereas SVN has a transactional log file, which only needs to be condensed, CVS, in contrast, maintains a file-level based log file, which does not enable a transactional view on the data by default. Thus, the transactions need to be reconstructed.

However, the conversion of a CVS log file into a transactional log file is not a huge effort, as developers typically check-in all changed files into the repository in one single commit (i.e., work step). Therefore, the developer and the commit message information of one transaction (commit) are the same (e.g., see [Breu and Zimmermann, 2006; Zimmermann and Weissgerber, 2004]). The date/time information can be (slightly) different, as every single file is uploaded separately and therefore gets its own time stamp. If a developer commits two large files, for example, the files may not have the same commit time information in the log entries.

Specifically, we take the complete CVS log file of a project and sort the entries by author, commit message, and date/time. Next, we combine the log elements with the same author and commit message and a given maximum time difference to one transaction (sliding window approach) similar to a SVN change log file (see Figure 5.1). A maximum time difference (time window) of up to five seconds between the log entries turned out to be an optimal value for all projects investigated. Finally, we assign a transaction number to all reconstructed transactions ordered by date and time to get a unique identification number for each transaction/commit.

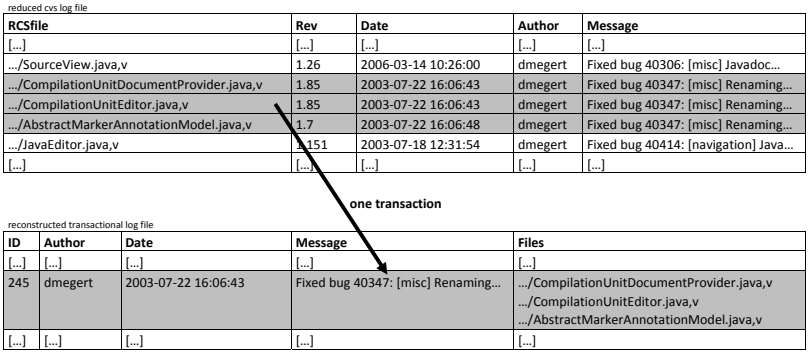


Figure 5.1: Reconstruction of the transactional change log file (simplified example for ECLIPSE)

5.4 Data Linking

As already mentioned in Section 3.3, BTSs and VCSs are, unfortunately, not automatically integrated. Therefore, the integration has to be established by scanning through the commit log messages for valid bug report numbers (see, e.g., [Fischer et al., 2003b]).

We enhanced this procedure by relaxing the bug number matching requirement and adding a time-window based verification. In other words, the process tries to match numbers used in commit/transaction messages with bug numbers. For all positive matches it then establishes whether the corresponding status of the bug report was changed to "fixed" in the period of seven days before or seven days after the relevant commit—a time period that seemed optimal for the projects we investigated (see discussion below). The result is a better linking ratio compared to previously presented approaches and datasets.

In the following sub-sections we define what fixed bug reports are and discuss our improved linking approach in detail.

5.4.1 Relevant Bug Reports

In OSS projects the bug reports maintained by BTSs are usually accessible to everyone (except for security relevant bug reports). In some cases a registration is needed, but there are typically no limitations on who is allowed to report a bug. As a result the quality of these data can vary as the databases can contain spam bug reports, duplicates, as well as feature requests camouflaged as reported bugs (see, e.g., [Bettenburg et al., 2007a] for more information about the quality of bug reports). To determine for instance the ratio at which bug reports of a certain project are linked to the commit log files, we have to find a definition for ***"fixed" bug reports***, which we define as follows:

DEFINITION: *Fixed bug reports are bug reports that have at least one associated fixing activity (status change to fixed) within the considered time period.*

We observe a ***fixing activity*** whenever a reported bug had its status changed to "fixed" at least once in its history. This change of status indicates that a developer performed some activity at some time to fix the reported bug.

5.4.2 Improved Linking Approach

It is critical for our work to identify the linked bug reports and the corresponding commits. We base our approach on the current technique of finding the links between a commit and a bug report by searching the commit log messages for valid bug report references. This technique has been widely adopted and is described in Fischer *et al.* [Fischer et al., 2003b] and used by other researchers [Moser et al., 2008; Śliwerski et al., 2005; Čubranić and Murphy, 2003; Čubranić et al., 2005; Zimmermann et al., 2007].

Our technique is based on this approach, but makes several changes to decrease the number of false-negative links (i.e., links that are valid references in the commit log but not recognized by the current approach). Specifically, we relaxed the pattern-matching to find more

potential mentions of bug reports in commit log messages and then verify these references more carefully in three steps (Steps 2–4):

1. We scan through the commit/transaction messages for numbers in a given format, or numbers in combination with a given set of keywords (see Table 5.1).
2. We exclude all false-positive numbers (e.g., release numbers, calendar dates, etc.) which have a defined format (see Table 5.2).
3. We check if the potential bug number exists in the BTS.
4. We check if the linked bug report is a fixed bug report and whether it has a fixing activity seven days before or seven days after the commit date (see Figure 5.2).

Table 5.1: Improved linking approach: Regular expressions to identify possible bug report numbers (step 1)

Category	Sample log message	Regular expression
"Bug report" keywords	"Reported problem in defect 23132 solved"	".*defect.*"
	"Bug 49115 - Cleanup on VMDisconnectException"	".*bug.*"
	"PR#1600: invoke_handler() doesn't handle mime arguments in content-type"	".*pr#.*"
	"PR:1056"	".*pr:.*"
"Bug fixing activity" keywords	"Fixing 115095. Removed stale nature."	".*fixing.*"
	"Fix for 114077"	".*fix.*"
	"fixed 105230: [BIDI] need to add nl/ to icon paths for reversed icons"	".*fixed.*"
	"closed 1620"	".*closed.*"
Numbers (in a specific format)	"#158227#: new file WW8FFData.cxx" or "#5543"	".*(#)[1-9][0-9]{2,5}#?.*"
	"#i91912# crash fixed; typo wrong array used..." or "#i56321"	".*(#i)[1-9][0-9]{2,5}#?.*"
	"[7098] & removed unneeded classes" or "Workaround for [19335] & [18997]"	".*\\[[1-9][0-9]{2,5}\\].*"
	"114891 Simple compare error in CoreUtility.java"	"^[1-9][0-9]{2,5}.*$"

Table 5.2: Improved linking approach: Regular expressions to exclude false-positive hits (step 2)

Category	Sample log message	Regular expression
Large numbers (more than 7 digits)	"file Q-build-report-200511091047.html added" or "tagged for v20030602" or "Build notes for I20030603" or "ZRH build input for 20050308-0800"	"[0-9]{7,}(-[0-9]{4})?"
Calendar dates	"updated doc 2003-01-03" or "about.html revised 2001/11/02"	"(200)[0-9]{1}[-./]{1}[0-9]{2}[-./]{1}[0-9]{2}"
	"New Board 17-09-2001" or "extrakt vom 24.10.2007"	"[0-9]{2}[-./]{1}[0-9]{2}[-./]{1}(200)[0-9]{1}"

In other words, the process tries to match numbers used in commit/transaction messages with bug numbers. To avoid false-positive numbers, the algorithm excludes numbers in a specific format such as calendar dates or large numbers (outside the number space of valid bug report numbers). For all positive matches it then established if the corresponding bug was fixed in the period of seven days before or seven days after the relevant commit—a time period that seemed optimal for the projects we investigated (see discussion below). With this improved process, we achieved higher recall than with other approaches. But is this higher recall a result of a higher false-positive rate and how many valid links have we left out (false-negatives)? We evaluate our linking algorithm and discuss the results in Section 5.5.2.

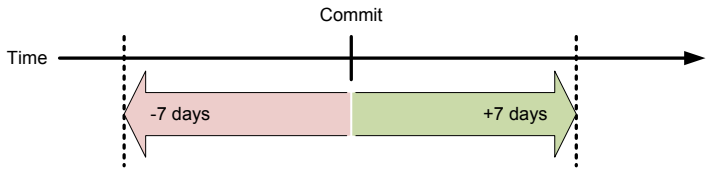


Figure 5.2: Valid time period for bug report links (existence of a fixing activity in this time period)

5.4.3 Bug Report Links: Valid or Not?

For our linking algorithm, we defined a valid time period of ± 7 days between the commit and the status change to the linked bug report. The difficult question in this context is the following: In which cases is a link valid and in which cases not? And why does a time constraint of ± 7 days seem to be optimum?

To discuss these questions, we first recall the bug fixing process (see Section 3.1.3). Assuming we have a BTS and a VCS, a developer would use these systems in the bug fixing process as follows:

1. Read the bug report and change the status to assigned (BTS)
2. Get the current version of program code, analyze the problem, fix the problem and commit the bug fix by declaring the bug report id in the commit message (VCS)
3. Change the status of the bug report to resolved with a resolution of fixed (BTS)

To check the validity of a link, we now use the time difference between step 2 and 3. Usually, a developer commits the bug fix to the VCS and, minutes to hours later, he also changes the status of the corresponding bug report in the BTS. If we now scan through the VCS log file and get links to fixed bugs whose status was change to fixed, for example, 180 days after the commit, we have to assume, that this is not a valid link. It is possible that the developer mistyped the bug report number or the number is just not a valid bug report number in this context (false-positive link, for instance affected by a release number).

On the other hand, developers tend to change the status of a bug report to fixed and commit additional bug fixes to the VCS hours to days later after changing the status but still refer to the already "fixed" bug. Even though this behavior is not desirable it is a common practice, particularly in CSS projects due to deadline/service level agreement restrictions.

The definition of a time period for valid links, therefore, is a trade-off between more links (fewer false-negatives) and a higher likelihood of invalid links (more false-positives). To get an optimal value for the valid time period, we ran our algorithm and linked all commits to bug

reports for which we found a valid bug report number (in particular, we left out step 4 of our linking algorithm). We then calculated the time differences between the commits and the status change (resolution="fixed") in the bug reports. The results for the OSS are shown in Figure 5.3. Defining a valid time window of $\pm$7 days, we considered almost 86% (ECLIPSE, NETBEANS, GNOME) and 68% (APACHE, MOZILLA) of all potential bug report links. Almost 77% (ECLIPSE, NETBEANS, GNOME) and 55% (APACHE, MOZILLA) of bug report links have a status change to the BTS in the previous 24 hours (0 days) after the commit. In the OPENOFFICE dataset we have high time difference values in both directions. Almost 49% of potential linked bug reports have a status change to fixed 20 days prior/after the commit. We discuss this quality issue later in this thesis in more detail. Unfortunately, we are not allowed to publish the results for the CSS projects. However, the values and distributions are in the same range but with a stronger tendency to higher values in a time difference between 0 and -7 days. Regarding the evaluation of our datasets for false-positives or false-negatives (see below), we were able to validate this time constrain range as optimum for all software projects investigated in this thesis.

Using an automatic approach to link the VCS with the BTS, indeed, we can only use heuristics to define which links are valid or not. But we are never sure if a given link is really valid. This is not possible without performing a manual examination of the specific changes to the program code in comparison to the bug report description. Unfortunately, such examination needs project-specific knowledge, skill, and plenty of time. Therefore, we are only able to perform such a verification for a subset of the datasets investigated (see Section 4.1.1). We discuss the verification procedure as well as the results in the next section.

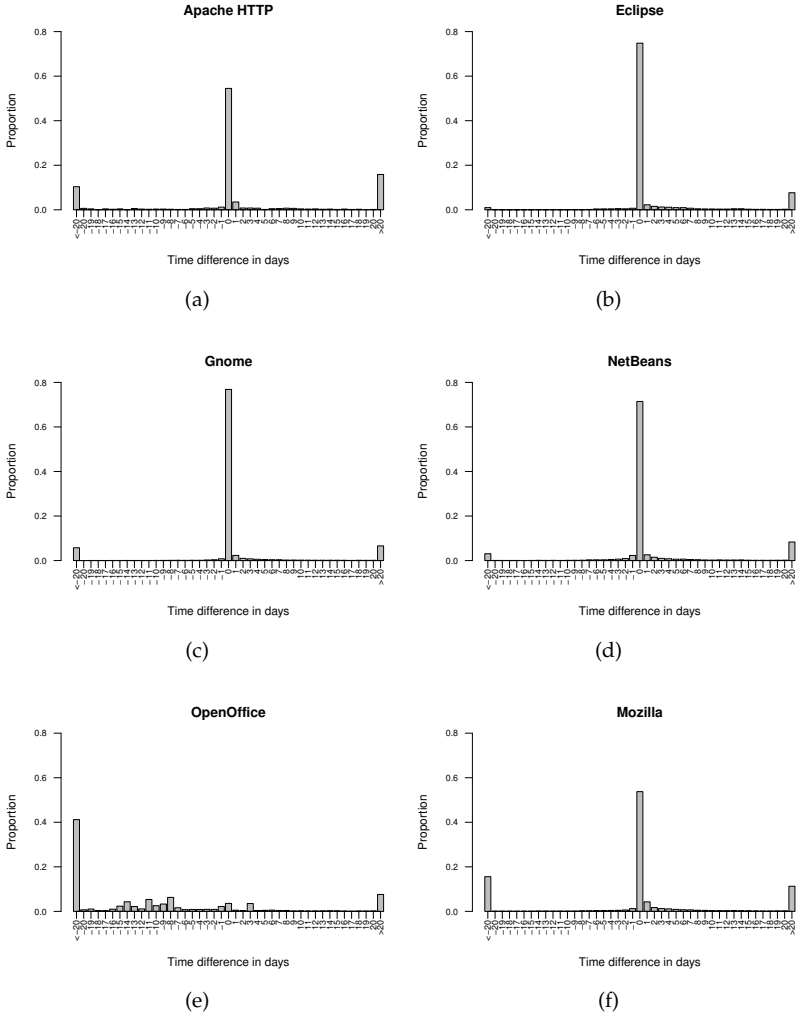


Figure 5.3: Time difference between commit and bug report status change (in days)

5.5 Evaluation of Our Data Extraction and Preparation Approach

5.5.1 Retrieving, Parsing, and Conversion Quality

The technique we use to retrieve, parse, and convert the data is comparable to previously presented approaches but differs in the amount of data we take into account.

First, we retrieve all the bug reports from the BTS, not only the ones mentioned in the VCS log file (which was done, for example, by Fischer *et al.* in [Fischer et al., 2003b]). This enables the possibility to evaluate the quality of the data as well as analyzing the datasets for bias.

Second, in addition to the static bug report information, we also retrieve the history (activity log) of every bug report. Therefore, we are able to verify the bug report numbers mentioned in the VCS log file as discussed in the previous section. In summary, we are able to counter the known issues (such as non-transaction oriented log files of CVS) and retrieve a complete picture of the reported bugs as well as logged changes to the program code, supporting Hypotheses 1.1 and 1.3.

But how can we verify the retrieving, parsing, and conversion functionality of our approach? Since we have no data manipulation (expect of the transactional log file reconstruction), we can compare the original data (stored in software engineering tools) with the parsed and converted data (stored in our relational database). Therefore, we focused our quality assurance activities in double checking and comparing the original data with the parsed data. Since we found no error, we assume there are no quality issues in this step of our approach.

5.5.2 Performance of the Linking Algorithm

Our linking approach differs from previously presented approaches in that it is less restricted to potential bug report numbers but veri-

fies them (see Section 5.4). The result is a better linking ratio compared to previously presented approaches and datasets. Specifically, we compared the ECLIPSE$_Z$ dataset provided by Zimmermann *et al.* (see [Dallmeier and Zimmermann, 2007; Zimmermann et al., 2007]) to our own ECLIPSE dataset. Under the same constraints, the technique we use identified more linked bugs than ECLIPSE$_Z$ (linking ratio of 33.05% compared to 16.03%; Table 5.3).

Table 5.3: Linking ratio: Comparison of ECLIPSE$_Z$ and ECLIPSE ("#" = number of)

Dataset	ECLIPSE$_Z$	ECLIPSE
Considered time period	2004-01-02 – 2006-04-08	2001-10-11 – 2008-02-28
#Bug reports	84 858	215 298
#Fixed bug reports	44 522	112 309
#Linked bug reports	7 137	37 122
#Commit messages	N/A	880 130
#Linked commit messages	7 137	185 040
#Linking ratio	16.03%	33.05%

But does this better linking ratio result in a higher false-positive rate and how many valid bug reports have we left out (false-negatives)? For our work, it is vital to obtain as faithful an extraction of all the linked commit messages as possible. Therefore, we manually inspected the results of our linking algorithm, looking for both false-positives and false-negatives in all datasets. The results of our manual inspection are shown in Table 5.4.

Specifically, for each dataset (see Table 5.4 - Column 1) we randomly selected a sample of commit log messages that were marked as linked messages to a given bug report. We then manually examined all the commit messages in our linked sample and verified the results of our linking algorithm for true-positives and false-positives, for example, year dates, release numbers, file names, etc. that were

recognized as valid bug report links. For all datasets we achieved a very low false-positive rate (including a low 95% confidence interval). Thus, our increased linking ratio is not a result of a (much) higher false-positive rate.

To verify our datasets for false-negatives, we also randomly selected a sample of commit log messages that were marked as unlinked. Again, we manually examined all the commit messages in our unlinked sample and verified the results of our linking algorithm for true-negatives and false-negatives . Although we found a few false-negative links (e.g., bug report numbers which are written with separators such as "Bug #223'344 fixed" or "Bug #22 33-44 fixed"), our manual verification showed that our linking algorithm works quite reasonably regarding the low false-negative rates (including the low 95% confidence intervals).

However, with our manual data examination, we were only able to evaluate our linking algorithm. Unfortunately, we are not able to verify if a specific commit really fixes the problem that was reported in the linked bug report. Therefore, we hired Justin Erenkrantz, an APACHE expert developer, to manually verify all the automatic established links in our APACHE evaluation sample dataset (Table 5.4 - Row 3 and 13). Justin found no false-positive links in our APACHE evaluation sample dataset, but found three commits that were not linked properly (false-negatives). Specifically, these links did not satisfy our heuristic for valid links (time constraint of $\pm$7 days between commit and status change to the bug report), and so were rejected as invalid links by our linking algorithm.

In summary, the extremely low levels of observed error (FP and FN rates) in our and Justin's manual examination does not pose a threat. Therefore, we assume that we are finding virtually all the commit log messages which the developers flagged as fixing specific bugs, supporting Hypothesis 1.3.

Table 5.4: Observed linking error in datasets ("#" = number of)

	#¬Linked	#¬Linked *verified*	#True-negatives (TN)	#False-negatives (FN)	FN rate (95% Confidence Interval)
APACHE complete	15 571	15 571 (100.00%)	15 558	13	0.000835 [0.000464, 0.001468]
APACHE evaluation	464	464 (100.00%)	461	3	0.006466 [0.001671, 0.020408]
ECLIPSE	154 991	1 500 (0.97%)	1 494	6	0.004000 [0.001627, 0.009153]
GNOME	655 668	124 374 (18.97%)	124 355	19	0.000153 [0.000095, 0.000243]
NETBEANS	326 445	74 219 (22.74%)	74 210	9	0.000121 [0.000059, 0.000239]
OPENOFFICE	93 384	93 384 (100.00%)	93 352	32	0.000343 [0.000238, 0.000490]
MOZILLA	131 909	32 900 (24.94%)	32 888	12	0.000365 [0.000198, 0.000657]
BSZKB#1	22 174	11 000 (49.61%)	10 996	4	0.000364 [0.000116, 0.001000]
BSZKB#2	21 944	21 944 (100.00%)	21 941	3	0.000137 [0.000035, 0.000436]

	#Linked	#Linked *verified*	#True-positives (TP)	#False-positives (FP)	FP rate (95% Confidence Interval)
APACHE complete	975	975 (100.00%)	975	0	0.000000 [0.000000, 0.004893]
APACHE evaluation	29	29 (100.00%)	29	0	0.000000 [0.000000, 0.145616]
ECLIPSE	66 165	30 000 (45.34%)	29 994	6	0.000200 [0.000081, 0.000459]
GNOME	52 692	10 374 (19.69%)	10 370	4	0.000386 [0.000124, 0.001060]
NETBEANS	51 839	18 769 (36.21%)	18 768	1	0.000053 [0.000003, 0.000346]
OPENOFFICE	13 326	13 326 (100.00%)	13 226	0	0.000000 [0.000000, 0.000359]
MOZILLA	88 551	17 724 (20.02%)	17 722	2	0.000113 [0.000020, 0.000455]
BSZKB#1	1 871	1 871 (100.00%)	1871	0	0.000000 [0.000000, 0.002554]
BSZKB#2	708	708 (100.00%)	708	0	0.000000 [0.000000, 0.006728]

5.6 Threats to Validity

Bugs Incognito. As mentioned in Section 5.1, we are not able to retrieve security relevant bugs since these are not available to the public. Therefore, some of the bug reports might be missing in our datasets. In addition, we have to assume that not all bugs are recorded in the project's BTS' as mentioned by Williams and Hollingsworth in [Williams and Hollingsworth, 2004]. Again, this may result in incomplete bug report data, or in other words: Some of the bugs may be incognito. In empirical software engineering most researchers commonly assume that all relevant bugs are recorded in the BTS. Therefore, we manually annotated our APACHE evaluation sample dataset and verify the completeness of the BTS bug report information (Chapter 8).

Missing Information. We implemented our toolset very carefully and checked the resulting datasets many times. Nonetheless, our approach and heuristics are, like previously presented approaches by other researchers, inexact and, therefore, we may have missed information stored in the software engineering tools considered.

Censored Data. For our linking algorithm we use the time difference between commit and bug report status change to verify bug report numbers in the VCS log for validity. An important issue in this context we should keep in mind is censored data: Every dataset that only uses a subset of the overall original data or uses continuous data will have missing information due to incomplete data (famous boundary problem). Since all software projects investigated are still under development, we may have such censoring issues. Let us assume we are interested in a bug report reported on December 30, 2009. Let us further assume that a developer commits the bug fix on December 31, 2009 and changes the bug report status (resolution="fixed") on January 1, 2010. If we now consider the time period between January 1, 2009 and December 31, 2009 only, our linking algorithm would reject a link between the VCS and BTS due to missing information about the status change on January 1st, 2010 (although our time constraint of ± 7 days is abided by). Whilst this problem may appear ar-

tificial, we will always have to consider the boundaries. We limit the error introduced by this censoring by purposefully employing data sets including very long periods of time (many years).

5.7 Concluding Discussion

In this chapter we presented in detail our approach for software engineering process data retrieval, processing, and linking. In particular, we introduced our approach to reconstruct a transactional log file from CVS change log data and presented our enhanced algorithm to link the VCS log data with the BTS. At the same time, we dealt with the question of which bug report links are valid and which ones are not. With our evaluation we showed that we achieve a higher linking ratio compared to other datasets and have a good linking quality at the same time (very low false-negative and false-positive rates). Therefore, we are able to support Hypotheses 1.1 and 1.2. In addition, compared to previously presented approaches, we retrieve all the BTS information including the activity log file for each bug report and, therefore, get a more complete picture of all bug fixing activities including all changes to a specific bug report, which supports Hypothesis 1.3.

Since we achieved a higher data as well as linking quality (in comparison to previously presented datasets), the datasets prepared with the technique presented in this chapter were used for several publications (e.g., [Bachmann and Bernstein, 2009a,b, 2010; Bachmann et al., 2010; Bird et al., 2009a; Rahman et al., 2010]).

Part III

Data Measurement and Evaluation

6

Data Quality and Characteristics Measurement[1]

How can we counter the known issues in preparing and linking software engineering process data? We answered this question in the previous part and, specifically, presented the software engineering projects investigated, the tools and processes used by these projects, and a step-by-step procedure for retrieving and preparing such data for further research. We also discussed several quality issues and compared our ECLIPSE dataset with the ECLIPSE_Z dataset provided by Zimmermann *et al.* based on the linking ratio (a quality measure). As discussed above, unfortunately, we have to assume that software datasets, even if we prepare them very carefully, are plagued by quality issues. In addition, due to non-uniformly used software engineering processes and tools, we may have vast differences in the characteristics of data across projects.

Although, software engineering process data is often and widely used in current research, little information is available about the qual-

[1] Major parts of this chapter have already been published [Bachmann and Bernstein, 2009b]

ity and the characteristics of these data. While a lot of literature concentrates on program code and software quality, only a few publications cover the quality of the software engineering process data (see Chapter 2). To our knowledge, none of the publications present a framework of measures to evaluate and compare software engineering process data and are, therefore, able to answer Research Question 2.

In order to overcome this knowledge gap, we introduce a framework of several process data quality and characteristics measures. With such quality measures, we are able to evaluate the data and process quality of a software project. In addition, these measures may indicate consequences for applications, which are based on such data. With characteristics measures, on the other hand, we are able to evaluate the the characteristics of a software project, the processes used and its data. The characteristics measures may be an indicator of the generalizability of research results that were developed on a specific dataset. In addition, we can use these measures to obtain a historical view on the quality and characteristics of the project's process data and overcome the knowledge gap of the quality and characteristics of software engineering datasets. Note that we are able to calculate these measures for every desired time period (only a few days up to several years), depending on our research goal. In addition, all measures are normalized and thus provide a good basis for the direct comparison of different software projects.

In the following sections, we introduce our framework of data quality and characteristics measures in detail. Specifically, we present measures to evaluate the data quality and characteristics of the BTS, VCS, and the combination (i.e., linking) of these two data sources. We describe each of the measures in detail and show how we calculate them.

6.1 Data Quality Measures

Evaluating the quality of a software system, currently published literature mainly focuses on quality measures such as the number of bugs related to lines of code (e.g., see [Basili et al., 1996; Binkley and Schach,

1998; Kan, 2002; Nagappan and Ball, 2005; Nagappan et al., 2006; Ohlsson and Alberg, 1996]). These software quality measures, however, do not take the quality of process data into account but measure the quality of the overall software system (in one dimension). In Chapter 11 we use a similar technique to evaluate the product (i.e., software) quality to calculate correlations between process data quality and product quality.

With our data quality measures, in contrast, we allow the quality of software engineering process data to be evaluated. Specifically, we define a number of measures which describe the quality of data in BTS, VCS, and its combination (i.e., linking).

6.1.1 Bug Tracking System Quality Measures

Ratio of Fixed Bug Reports

For linking of the BTS and VCS we are dependent on fixed bug reports (see Section 5.4) as we should have an associated bug fix commit mentioned in the VCS log file (see [Śliwerski et al., 2005]). The fewer the reported bugs have a resolution of "fixed", the fewer the bug reports were fixed by developers. All other bug reports are not relevant to track changes to the program code. The ratio of fixed bug reports, therefore, represents the usefulness of information in the BTS for tracking program code changes.

$$DQ_{rfb} = \frac{\text{\#fixed bug reports}}{\text{\#bug reports}} \tag{6.1}$$

Ratio of Duplicate Bug Reports (All Bug Reports)

Duplicates occur especially in projects with a large user community and a lot of inexperienced bug reporters. Inexperienced bug reporters are often unable to find similar bug reports due to a lack of search skills, or are not interested in finding a similar bug report. Thus, much more time is needed to assign bug reports to developers and identifying/eliminating duplicates (see [Anvik et al., 2006; Hooimeijer and

Weimer, 2007; Ko et al., 2006]). On the other hand, duplicate bug reports may contain additional information on a specific problem and can, therefore, help to resolve bugs more quickly, as described by Bettenburg *et al.* (see [Bettenburg et al., 2008]). Nonetheless, duplicates may affect serious problems in bug fixing processes if they are not recognized properly (e.g., more than one developer tries to fix the same problem without knowing that another has already done so).

$$DQ_{rdb} = \frac{\text{\#duplicate bug reports}}{\text{\#bug reports}} \tag{6.2}$$

Ratio of Works-for-me and Invalid Bug Reports (All Bug Reports)

Reported bugs that cannot be reproduced by developers and for those based on the bug description "no error was found" usually get the status "works-for-me" or "invalid". There are various reasons for such bug reports: User-site specific problems, too short a problem description, problems in understanding the description, important problem description information is missing (e.g., steps to reproduce), etc. According to Bettenburg *et al.* [Bettenburg et al., 2007a,b], a good bug report has specific criteria to fulfill. Because invalid bug reports, even if they describe real problems, are mostly useless, they cost much time for developers and are, therefore, undesired.

$$DQ_{rib} = \frac{\text{\#works-for-me bug reports + \#invalid bug reports}}{\text{\#bug reports}} \tag{6.3}$$

6.1.2 Version Control System Quality Measures

Ratio of Empty Commit Messages

All program code changes should be traceable and justified. This enables the possibility of understanding the history and evolution of the program code, for example, during bug fixing activities, and supports

new developers in understanding the program code. In addition, unjustified program code changes may increase operational risks and, as a result, requirements by laws and regulations for the justification and traceability of all program code changes are becoming more and more common (see Chapter 12 for a full discussion). Empty messages, unfortunately, contain no information about the commit's reason (justification) and are, therefore, undesired. The more empty messages exist, the more information is lost. Therefore, this value should be as low as possible.

$$DQ_{rem} = \frac{\text{\#empty commit messages}}{\text{\#bug reports}} \tag{6.4}$$

Ratio of Commit Messages with Bug Report Links (Excluding Empty Commit Messages)

Program code changes should be traceable and justified as discussed above (i.e., the reason for the change should be discernible). Possible justifications are a bug fixing activity or the implementation of a new feature. Referring a bug report by its bug report number is one of the possible ways to justify a commit. Therefore, the higher this ratio, the more program code changes are motivated by a bug report (and are therefore justified).

$$DQ_{rlm} = \frac{\text{\#commit messages with bug report links}}{\text{\#commit messages (w/o empty)}} \tag{6.5}$$

6.1.3 Combined Quality Measures

Ratio of Linked Bug Reports (All Bug Reports / Fixed Bug Reports Only)

The ratio of linked bug reports is one of the most important quality measures. It shows how well a BTS is linked in the VCS log file. With our enhanced linking approach we are able to increase this ratio compared to previously presented approaches/datasets (see Section 5.5.2).

A poor linking ratio may have two reasons: First, our linking algorithm does not perform well or, second, even if our linking algorithm finds virtually all the commit messages a developer has marked as bug fixes, we still have a low ratio due to missing information in the VCS log file. Independently of the reasons for a poor link ratio, we have missing information in such datasets which may have effects on research results (e.g., bug prediction research). A high linking ratio, therefore, is highly desired.

Specifically, we use two versions of this quality measure: First, we put the number of linked bug reports into perspective with the total number of bugs reported (Eq 6.6). Second, we qualify the number of linked bugs in comparison to the number of fixed bugs only (Eq 6.7).

$$DQ_{rlb_{all}} = \frac{\text{\#linked bug reports}}{\text{\#bug reports}} \tag{6.6}$$

$$DQ_{rlb_{fixed}} = \frac{\text{\#linked bug reports}}{\text{\#fixed bug reports}} \tag{6.7}$$

6.2 Data Characteristics Measures

Usually, projects are classified based on simple statistical measures such as the number of reported bugs (see Table 4.1). With our data characteristics measures we now take additional information such as BTS activities into account and combine the data sources to get a deeper insight into the characteristics of software project process data. Based on such data characteristics measures, we are able to compare the projects and uncover possible threats in the generalizability of results across projects. We hypothesize (Hypothesis 2.3) that we have vast differences in data characteristics across projects due to non-uniformly used software engineering processes and tools.

6.2.1 Bug Tracking System Characteristics Measures

Average Number of Status Changes per Bug Report

Status changes in bug reports are a good indicator of the degree of collaboration and the existence of a (well) defined bug fixing process (see Section 3.1.3 and [Hooimeijer and Weimer, 2007]). In most projects a status model defines which states a bug report has to pass through (bug life cycle) and, therefore, each bug report should have at least three associated status changes: New – Resolved [Fixed] – Verified – Closed.

$$DC_{asb} = \frac{\text{\#bug report status changes}}{\text{\#bug reports}} \tag{6.8}$$

Average Number of Comments per Bug Report

The number of comments is a good indicator of the activity of a community, since many projects use the BTS to discuss reported problems and possible solutions. On the other hand, many comments can also be an indicator of insufficient bug descriptions (see [Anvik et al., 2006; Hooimeijer and Weimer, 2007]). Bug reports with a lot of comments, therefore, should be analyzed in more detail.

$$DC_{amb} = \frac{\text{\#bug report comments}}{\text{\#bug reports}} \tag{6.9}$$

Average Number of Attachments per Bug Report

Depending on the software product, attachments such as screenshots, code snippets, or input/output files are very helpful to understand and fix a problem. Therefore, experienced users usually add such information to a bug report, so that attachments are usually a sign of a good bug report (see [Anvik et al., 2006; Bettenburg et al., 2007b; Hooimeijer and Weimer, 2007]). However, adding screenshots or input/output files is only feasible for software systems that have a GUI

or provide import/export functionality. The absence of attachments may therefore be a result of the project's characteristics (no GUI, import/export functionality, etc.) rather than a sign of poor quality.

$$DC_{aab} = \frac{\text{\#bug report attachments}}{\text{\#bug reports}} \tag{6.10}$$

Average Number of Bug Reports per Bug Reporter

The more bugs a user reports, the higher the quality of reports typically is. This is a result of learning effects: With every additional report, the user becomes faster and describes a specific problem more precisely because he knows what information a developer needs to know to fix a problem (i.e., bug). Better bug reports allow efficient bug fixing (see [Bettenburg et al., 2007a]). Therefore, the more bug reports a bug reporter, the better the quality of these reports and the faster such bugs can usually be fixed. In addition, this measure indicates whether a project performs professionalized testing or testing by users (see Section 3.1.2). If a project performs professionalized testing, usually only a few testing experts report all the bugs and, therefore, we have high values. Performing user testing, on the other hand, usually results in many users that report only a few bugs.

$$DC_{abr} = \frac{\text{\#bug reports}}{\text{\#bug reporters}} \tag{6.11}$$

6.2.2 Version Control System Characteristics Measures

Average Length of Commit Messages (Excluding Empty Commit Messages)

During the check-in process, conscious developers usually write a commit message which contains the reason (i.e., justification) for the commit. Therefore, we are able to track all program code changes, which is

in any event good practice (see discussion above). Such commit messages usually contain referencing numbers to other tools such as BTSs or requirement databases and are limited to a few characters. In some projects, developers not only submit the changes to the program code to the VCS but also use the new version of the code as the commit message, which is, however, comparable to an empty message since we have no information about the commit's reason. Very large messages may therefore be an indicator of useless information in the commit messages (quantity instead of quality). This has a similar consequence as empty messages (the commit messages contain no justification and are therefore undesired). However, the average length of commit messages is a good hint of how a VCS is used by a project.

$$DC_{alm} = \frac{\sum \text{length(commit messages)}}{\text{\#commit messages (w/o empty)}} \tag{6.12}$$

6.2.3 Combined Characteristics Measures

Average Number of Bug Report Status Changers per Developer

The relation between bug report status changers (people that are allowed to change the status of a bug report) and developers shows how open the access to the BTS is defined. In the ideal case, a developer fixes a bug and changes the resolution of the bug report to "fixed". Later, someone verifies (i.e., tests) the bug fix and, finally, someone closes the bug report. Using this process, a maximum of three different people touch each bug report. Because bug reports sometimes have to be re-opened or re-tested again, some additional status changes might be needed (e.g., by additional people). The number of people that change the status in relation to the number of developers is now a good indicator of how many people acting in a project are not developers (e.g., testing experts, quality assurance experts, development directors).

$$DC_{asd} = \frac{\text{\#bug report status changers}}{\text{\#developers}} \tag{6.13}$$

Average Number of Bug Reporters per Developer

As already discussed, the way a project is used to test new releases has major impacts on the data quality and characteristics. If we have professionalized testing, the relation between bug reporters and developers is usually small and on a constant level. If we have user testing, we usually have many more bug reporters and the larger a user community is, the more reporters exist in relation to developers. A higher value of this measure is not an indicator of poor data quality *per se*. But the fewer the reporters in relation to the developers, the greater the likelihood of the developers giving personal feedback in the case of poor bug reports to ensure better bug reports in the future. Consequently, small user communities can profit from personal relations between users and developers, which may in turn lead to more efficient bug handling. Therefore, this measure is, again, an indicator of the method of testing and the size of a user community that reports bugs.

$$DC_{ard} = \frac{\text{\#bug reporters}}{\text{\#developers}} \tag{6.14}$$

Average Number of Bug Reports per Developer

The more bug reports a developer has touched, the greater his experience, and experienced developers are very valuable for a project. Therefore, this value is an indicator of the average experience of the developer in a project (we use this measure later in this thesis to evaluate bug feature bias). On the other hand, this measure is also an indicator of the average workload of the developers.

$$DC_{abd} = \frac{\text{\#bug reports}}{\text{\#developers}} \tag{6.15}$$

Average Number of Fixed Bug Reports per Developer

The more bug reports a developer has fixed, the higher his experience usually is. Hence, this measure is also an indicator of the experience

mixture of the developers of a project.

$$DC_{afd} = \frac{\text{\#fixed bug reports}}{\text{\#developers}} \tag{6.16}$$

Average Number of Bug Report Links per Linked Bug Report

Usually, a developer fixes a bug and commits the bug fix in one single work step. Therefore, a bug report is usually mentioned in the VCS log file only once. On the other hand, major bugs may require several commits to the same problem and if bugs have to be re-opened several commits may occur as well. With this measure, therefore, we evaluate how often a bug report was mentioned by developers in the commit log. High values of this measure indicate that a project often has major bugs that need multiple commits or that we have developers undertaking unusual behavior in the project (e.g., end-of-day commits for backup purposes; Section 7.1)

$$DC_{alb_{linked}} = \frac{\text{\#bug report links}}{\text{\#linked bug reports}} \tag{6.17}$$

Average Number of Commits per Bug Report (All Bug Reports / Fixed Bug Reports Only)

The relation of commits and bug reports offers an indication of how much a project is driven by bug fixes in relation to new feature development or refactoring tasks. The lower this value, the higher the proportion of bug reports, which indicates that a project is in a maintenance phase rather than in a phase of implementing a new release. Unfortunately, the problem of end-of-day commits (see Section 7.1) falsifies the information of this value.

Specifically, we use two versions of this characteristics measure: First, we put the number of commits into perspective with the total number of bugs reported (Eq 6.18). Second, we qualify the number of

commits in comparison to the number of fixed bugs only (Eq 6.19).

$$DC_{acb_{all}} = \frac{\text{\#commits}}{\text{\#bug reports}} \tag{6.18}$$

$$DC_{acb_{fixed}} = \frac{\text{\#commits}}{\text{\#bug reports}} \tag{6.19}$$

Average Number of Commits per Developer

A common measure to evaluate the developer's experience is its number of fixed bugs. Calculating the relation between number of commits and developers allows us to evaluate, indirectly, the average experience of the developers in terms of their activity in a project. Again, end-of-day commits may compromise comparisons of the calculated values across projects.

$$DC_{acd} = \frac{\text{\#commits}}{\text{\#developers}} \tag{6.20}$$

6.3 Threats to Validity

Data Outliers. For our data characteristics measures we calculate the mean of the values instead of the median. Note that the median is much more representative of the central tendency of a dataset. In addition, outliers can dramatically impact the mean, whereas the median is less affected. Note also that the robustness of the median against outliers is not a quality criteria but lies in the nature of this measure. Specifically, the median is less "disturbed" by outliers but also reports them quite poorly. However, it depends on the data whether the mean or median actually gives us a more accurate reflection of an "average" value and supports the interpretation of results best. Since we believe that outliers in the data are "valid" and "important" to compare the measures across projects, we decided to use the mean instead of the median. However, in some of the measures, we explicitly exclude unwanted outliers such as empty messages so as to not falsify the values.

Nonetheless, we may have threats due to issues regarding "invalid" outliers in the data.

Data Quality Issues. Regarding Hypothesis 2.2, we believe that software engineering datasets are plagued by data quality issues such as missing information (i.e., data). We defined our measures to uncover such quality issues, allowing an evaluation across projects. We therefore tried to define measures that are robust against such issues itself. Nonetheless, some of our measures, especially our data characteristics measures, may be affected by quality issues such as incomplete data in the BTS or end-of-day commits that influence the results. However, the data characteristics measures should uncover such "data characteristics" and not qualify the data.

Definition of a Developer. In our framework of data quality and characteristics measures we defined a developer as a person who is allowed to commit program code changes to the VCS of a project. We acknowledge possible threats to validity since in many OSS projects only core developers are allowed to submit to the VCS although many other "non-core" developers may contribute to a project. Hence, a bug fix may be "developed" and submitted to the BTS as comment or attachment but submitted to the VCS by a core developer who successfully verified the bug fix. Therefore, many more developers may contribute to a project than identifiable in the VCS.

6.4 Concluding Discussion

In this chapter we presented a framework of data quality and characteristics measures for software engineering processes. This framework enables the comparison of different projects in terms of their process data quality and characteristics. We described every measure in detail and showed how to calculate them.

In the next chapter, we evaluate and compare the datasets investigated in this thesis using the measures introduced and test hypotheses related to Research Question 2.

Later in this thesis, we use these measures again to calculate cor-

relation values between the measures (Chapter 10) and between the measures and number of bugs as a product quality measure (Chapter 11).

7

Software Engineering Data Evaluation[1]

In the previous chapter, we introduced a framework of twenty data quality and characteristics measures. According to these measures, we now test the following hypotheses related to our Research Question 2:

HYPOTHESIS 2.1: *Our framework of data measures can evaluate and compare the data characteristics and quality across several software projects.*

HYPOTHESIS 2.2: *Software engineering datasets are plagued by data quality issues such as missing information.*

HYPOTHESIS 2.3: *Software engineering datasets vary in their characteristics across projects, especially between open source and closed source software projects.*

To test the hypotheses, we calculate these measures for all projects investigated. The results allow us to better understand the data and

[1]Some of the results have already been published [Bachmann and Bernstein, 2009b]

projects for further research and provide an indication of the characteristics of these projects and the processes. Hence, the results provide some indication with respect to the generalizability of findings by drawing on process data of one of the projects.

The evaluated process data quality measure values for all projects are shown in Table 7.1, while Tables 7.2 and 7.3 list the characteristics measure values. Please note that all values are calculated based on the considered time periods only (see Table 4.1).

In the following sections we discuss some of the values with respect to (i) the difference between OSS and CSS, (ii) bug report links, (iii) the ratio between bug reporters and developers, (iv) the bug status changes, and (v) the change log quality and characteristics.

7.1 Comparison of Open Source and Closed Source Data

OSS and CSS projects usually make use of different software engineering processes and use software tools differently (see Chapter 3). Data characteristics, therefore, may vary across projects—especially between OSS and CSS projects. In this section, we compare the data quality and characteristics between OSS and CSS projects and analyze Hypothesis 2.3.

Considering the CSS datasets from Zurich Cantonal Bank (ZKB) (BSZKB#1 and BSZKB#2), there are some values that stand out compared to the OSS projects: First, we have many more commits in relation to bug reports ($DC_{acb_{all}}$ and $DC_{acb_{fixed}}$) and developers (DC_{acd}). Second, compared to most OSS projects, we have fewer commit messages which contain bug report links in relation to all commit messages (DQ_{rlm}). In interviews with the project management from ZKB we found that end-of-day commits are one reason for these comparably high values: At ZKB, developers usually check-in their entire work at the end of the day into the repository, using it, against its intended use, as a backup system. In both ZKB projects we have a small

Table 7.1: Evaluated process data quality

Quality measure	APACHE	ECLIPSE	GNOME	NETBEANS
Ratio of fixed bug reports, DQ_{rfb}	28.80%	52.16%	26.40%	52.41%
Ratio of duplicate bug reports (all bug reports), DQ_{rdb}	12.39%	13.03%	33.56%	14.82%
Ratio of works-for-me & invalid bug reports (all bug reports), DQ_{rib}	34.46%	12.74%	4.32%	12.91%
Ratio of empty commit messages, DQ_{rem}	0.02%	20.17%	1.27%	0.50%
Ratio of commit messages with bug report links (w/o empty), DQ_{rlm}	4.9%	34.37%	7.73%	12.92%
Ratio of linked bug reports (all bug reports), $DQ_{rlb_{all}}$	12.51%	17.24%	10.29%	28.62%
Ratio of linked bug reports (fixed bug reports only), $DQ_{rlb_{fixed}}$	43.43%	33.05%	38.99%	54.60%

Quality measure	OPENOFFICE	MOZILLA	BSZKB#1	BSZKB#2
Ratio of fixed bug reports, DQ_{rfb}	38.93%	31.93%	56.73%	47.66%
Ratio of duplicate bug reports (all bug reports), DQ_{rdb}	16.12%	26.02%	1.38%	0.16%
Ratio of works-for-me & invalid bug reports (all bug reports), DQ_{rib}	19.46%	22.16%	0.09%	24.06%
Ratio of empty commit messages, DQ_{rem}	0.03%	0.22%	9.61%	39.02%
Ratio of commit messages with bug report links (w/o empty), DQ_{rlm}	9.11%	40.26%	8.61%	5.13%
Ratio of linked bug reports (all bug reports), $DQ_{rlb_{all}}$	2.89%	11.13%	21.17%	17.03%
Ratio of linked bug reports (fixed bug reports only), $DQ_{rlb_{fixed}}$	7.43%	34.87%	37.31%	35.74%

Table 7.2: Evaluated process data characteristics (part 1)

Characteristics measure	APACHE	ECLIPSE	GNOME	NETBEANS
Average status changes per bug report, DC_{asb}	1.77	1.83	1.35	1.97
Average comments per bug report, DC_{amb}	3.58	4.32	2.95	4.46
Average attachments per bug report, DC_{aab}	0.32	0.41	N/A	0.47
Average bug reports per bug reporter, DC_{abr}	1.42	11.43	2.71	11.17
Average length of commit messages (w/o empty), DC_{alm}	102.44	44.52	174.88	59.38
Average number of bug report status changers per developer, DC_{asd}	12.27	27.94	7.54	3.90
Average bug reporters per developer, DC_{ard}	46.80	100.73	105.50	17.61
Average bug reports per developer, DC_{abd}	66.63	1151.33	285.50	196.64
Average fixed bug reports per developer, DC_{afd}	19.19	600.58	75.38	103.06
Average bug report links per linked bug report, $DC_{alb_{linked}}$	1.38	1.73	1.21	1.42
Average commits per bug report (all bug reports), $DC_{acb_{all}}$	3.83	6.56	4.60	7.25
Average commits per bug report (fixed bug reports only), $DC_{acb_{fixed}}$	13.31	12.58	17.42	13.83
Average commits per developer, DC_{acd}	5.46	74.99	12.45	80.96

Table 7.3: Evaluated process data characteristics (part2)

Characteristics measure	OpenOffice	Mozilla	BSZKB#1	BSZKB#2
Average status changes per bug report, DC_{asb}	2.93	1.82	7.74	5.22
Average comments per bug report, DC_{amb}	6.53	7.41	N/A	N/A
Average attachments per bug report, DC_{aab}	0.60	0.79	1.10	0.33
Average bug reports per bug reporter, DC_{abr}	4.51	4.06	58.97	16.51
Average length of commit messages (w/o empty), DC_{alm}	75.65	96.00	53.88	43.61
Average number of bug report status changers per developer, DC_{asd}	29.10	34.42	4.35	0.80
Average bug reporters per developer, DC_{ard}	161.53	187.90	2.61	0.80
Average bug reports per developer, DC_{abd}	728.17	761.88	153.78	13.06
Average fixed bug reports per developer, DC_{afd}	283.49	243.30	87.24	6.22
Average bug report links per linked bug report, $DC_{alb_{linked}}$	5.22	1.74	1.26	3.43
Average commits per bug report (all bug reports), $DC_{acb_{all}}$	7.71	0.44	14.55	35.39
Average commits per bug report (fixed bug reports only), $DC_{acb_{fixed}}$	19.81	1.39	25.65	74.27
Average commits per developer, DC_{acd}	34.76	338.65	857.96	462.29

number of developers who all work full-time on the development of the specific project. This is also a reason for the comparably very high DC_{acd} values as there are only a few, but very active developers.

Regarding the BTS data characteristics measures, further differences between the ZKB and the OSS projects stand out. For both ZKB projects, we have many more status changes to the bug reports (DC_{asb}), comments on bug reports are not present (DC_{amb}), and every bug reporter reports on average many more bugs (DC_{abr}). The reason lies in the well-defined bug handling process followed within ZKB that leads to many status changes for each bug report. The values of the other two measures have a similar explanation: Testing at ZKB is performed by a few professional testing engineers. This is in contrast to testing in OSS, where testing by users is usually performed. Therefore, every bug reporter within ZKB reports almost 60 bugs (BSZKB#1) or 17 bugs (BSZKB#2), including a rich description of the bug. Regarding the number of attachments added to bug reports on average (DC_{aab}), we see that in the BSZKB#1 project every bug report has at least one attachment but the BSZKB#2 has a much lower value. Again, we asked the project management at ZKB to explain this finding and found a simple reason: The BSZKB#1 project has a user GUI whereas the BSZKB#2 project does not provide any user interaction possibility but provides interfaces to other software components. Therefore, testing experts of the BSZKB#1 project usually add screenshots and input/output files to describe the unwanted behavior in more detail, whereas BSZKB#2 testing experts, in contrast, are usually able to describe a problem well without any additional information in attachments.

In OSS projects, testing is usually performed by users. However, most users only report a few bugs in their lifetime and are, therefore, inexperienced in using a BTS and finding similar bugs (see the low levels of DC_{abr} in most OSS projects). Therefore, most OSS projects have a very high ratio of "duplicate" bug reports in their BTS (DQ_{rdb}). In GNOME, for instance, almost 34% of all reported bugs are duplicates. In contrast, "duplicate" bug reports are almost non-existent in the ZKB projects. There are two reasons for this: First, the BTS is not publicly

available; only registered testing experts are allowed to report bugs. Second, the testing activities are quite well-planned and follow defined test steps/cases (including test data). Therefore, we have almost no overlapping in testing in contrast to user testing, where software tests are usually undertaken in an uncoordinated manner (try-and-error approach). In addition, in the BSZKB#1 project, "works-for-me" and "invalid" bugs are also almost non-existent (DQ_{rib}) since we have professionalized testing by experts who know exactly how a good bug report should look like. However, the BSZKB#2 project has a much higher value. Asking the BSZKB#2 project management, we found that this is not affected by poor bug report quality, but rather by bug reports that were rejected/forwarded to other ZKB projects. We found that the impact-of-defect vs. cause-of-defect problem often appears in reporting bugs in this project (see Section 8.2.3 for a full discussion).

Consequently we can summarize that the rigorously followed procedures in the ZKB projects are well-reflected in the process quality and characteristics measures, supporting Hypothesis 2.1. In addition, we showed that there are vast differences in data characteristics across projects, especially between OSS and CSS projects (Hypothesis 2.3).

7.2 Ratio of Linked Bug Reports

VCSs and BTSs contain huge amounts of valuable data for empirical software engineering. However, the integration (i.e., linking) of these two data sources provides even more valuable information and enables promising research possibilities such as defect prediction. Therefore, the linking of data is of major importance.

Presenting our extended linking algorithm in Section 5.4, we already compared the linking ratio between our ECLIPSE dataset and the ECLIPSE_Z dataset provided by Zimmermann *et al.* showing that we achieve a much higher linking ratio. We also verified our dataset and conclude, according to the very low levels of error (Table 5.4), that we are finding virtually all the commits a developer marked as a specific bug fix. Nonetheless, we hypothesize that the data stored in

software engineering tools is plagued by data quality issues such as missing (linking) information (Hypothesis 2.2). Therefore, we discuss in this section the evaluated linking ratio ($DQ_{rlb_{all}}$ and $DQ_{rlb_{fixed}}$) for all projects investigated.

Unfortunately, the linking ratio values for all of our projects are rather sobering: In the best case (when considering fixed bug reports only), we only have roughly 55% (NETBEANS) of all bug reports linked in the VCS log file. On average, these values are even lower. The poor linking ratio (in all of our analyzed projects) is a strong indicator of the missing traceability and justification of program code changes. In addition, it is questionable whether the sample of linked (and fixed) bug reports is a representative sample of all fixed bugs. This may lead to bias in software engineering datasets, and bias may increase problems in research results that are based on such data. Therefore, we analyze and discuss these issues in more detail in Part IV.

7.3 Proportion of Bug Report(er)s and Developers

An important factor for the bug fixing efficiency of a developer is the quality of a bug report. Is all information available that a developer needs to know? Is the problem description clear enough? Bettenburg *et al.* evaluated the quality of bug reports to answer these questions among others and found vast differences in the quality of bug reports [Bettenburg et al., 2007a,b]. They proposed enhancing the quality of bug reports by improving the functionality of BTSs, for example, by analyzing the quality of the bug whilst typing.

However, in some of the projects investigated such enhanced BTSs are not necessary since we have only a few people reporting bugs. The ZKB projects, for instance, have very low proportions between bug reporters and developers (DC_{ard}): In BSZKB#1 there are only 2.61 bug reporters per developer and in BSZKB#2 there are, interestingly, actually more developers than bug reporters ($DC_{ard} = 0.80$). For the OSS

projects, in contrast, values above 100 are no exception. The small numbers in the CSS projects open up another possibility for improving the bug report quality: A developer simply has to write an email or pick up the phone and tell the bug reporter how a good bug report should look like. This is a normal procedure in the ZKB projects. The lower ratios of duplicate and invalid bug reports (DQ_{rdb} and DQ_{rib}) are an indicator of the higher quality of bug reports by such briefing and training. However, only the small proportions make such a briefing possible at reasonable costs. Since in most OSS projects much higher values are found, it is not feasible to tell every bug reporter how a good bug report should look like or provide personal feedback. In ECLIPSE, for instance, there are more than 100 bug reporters and more than 600 fixed bug reports per developer.

Consequently, these measures can serve as a proxy of how easy the quality of bug reports can be handled and improved by developers and in which projects a BTS extension (as proposed by Bettenburg *et al.*) would be valuable in assisting bug reporters in writing good bug reports.

7.4 Bug Report Status Changers

In almost all OSS projects there are a large number of people who change the status of a bug report (bug report status changers) in proportion to the number of developers (DC_{asd}). This is quite surprising. Remember the bug fixing process discussed in Section 3.1.3: A developer usually fixes a bug and changes the resolution of the bug report to fixed. Later, someone re-checks (i.e., verifies) the bug fix, and finally, someone closes the bug report (status = closed). Sometimes a bug report has to be re-opened. In particular, it is odd to observe that in the OSS projects the status of a bug report changes very rarely (only about twice, see DC_{asb}) even though the number of status changers is at least an order of magnitude larger than the number of developers. Consequently, we theorize that in these projects a large number of users are allowed to change a bug report status, which we find slightly

problematic.

In the CSS projects as well as in NETBEANS the values are much smaller—probably due to a rigorous bug fixing procedure. Given that the bug fixing process is usually managed using the bug status changes we believe that only a defined number of trusted users should have the permission to change the status of a bug report.

7.5 Version Control System Log File Revised

There are various reasons why program code changes should be traceable and justified (see Section 6.1.2), although this is not an easy task. In the meantime, most medium- to large-scale OSS as well as CSS projects make use of a VCS where information about the date, time, author, and the changed content is stored. Therefore, such changes are mostly traceable. The rationale or justification for the changes, on the other hand, is often difficult to reconstruct due to missing or inconclusive statements in the VCS log messages. In the ECLIPSE project, for instance, more than 20% of all commit messages are empty (DQ_{rem}) and in the BSZKB#2 project, even worse, almost 40% of all commit messages. As already discussed, empty commit messages contain no information about the reason for a commit. Regarding the ratio of empty commit messages and the low levels of linked bug reports (roughly 55% of all bug reports linked, see Section 7.2), we have to conclude that most program code changes are not justified.

However, according to Klein, every commit should include a statement about change rationale [Klein, 1993]. Possible categories could be:

- Implementing a new (requested) feature
- Fixing a bug
- Refactoring tasks

Newly added features could be substantiated by a link to a feature

or project database. The same should be done for bug fixing commits. For refactoring tasks there should be at least a statement of what was refactored and why. Modern repository systems like JAZZ enforce rationale capture by allowing commits only in combination with a developer task (such as a bug fix or a new feature task). Therefore all program code changes should be justified and documented.

Nonetheless, the usage of such tools or the enforcement of rules to justify all commits has to be enabled by software engineering practitioners. In Part V, therefore, we discuss why practitioners should do so.

7.6 Concluding Discussion

In this chapter we used our framework of data quality and characteristics measures to evaluate the data of our investigated projects. We found, not surprisingly, that all projects—both OSS and CSS—are plagued by data quality issues. Specifically, we showed how badly bug reports are linked to commits, such that all datasets have a linking ratio on a poor level of below 55%. In the worst case (OPENOFFICE) only 7.43% of all fixed bug reports are properly mentioned in the VCS. This may increase the number serious problems in empirical software engineering results, since the linking of the BTS and VCS is critical for most research results. As only a sample of the fixed bug reports are mentioned in the VCS log file, we have to analyze whether this sample is representative of all fixed bug reports or not. If not, our datasets may be biased (even though we prepared the data very carefully) and the research results affected.

In addition, we used our data characteristics measures to evaluate the data characteristics across all projects investigated and found, again not surprisingly, vast differences across all projects—especially between OSS and CSS projects. This increases the risk of problems in the generalization of research results.

Summarizing, our framework of data quality and characteristics measures presented in the previous section allow a fast and compre-

hensive overview of software engineering datasets. Therefore we can support Hypothesis 2.1. Regarding the evaluated data quality measures, we have shown that all datasets are plagued by data quality issues, while the low levels in the ratio of linked bug reports is of main interest. Unfortunately, the poor linking ratio is a result of missing information in the VCS log file (e.g., developers do not properly refer to a bug report number during the check-in process of a bug fix) rather than a problem with our linking technique. In addition, we have shown that differences in software engineering processes (e.g., the way to test a software project) led to vast differences in data characteristics. Therefore, we are able to support Hypotheses 2.2 and 2.3.

In this chapter we evaluated the data quality and characteristics of all projects investigated. But is there an effect on (i) results of empirical software engineering studies or (ii) the work of practitioners as hypothesized? In the next two parts, we analyze the influence of poor data quality on research results as well as on product and process quality.

Part IV

Why Should Empirical Software Engineering Researchers Care about Data Quality in Software Engineering?

8

Bugs Incognito and Commits Incognito[1]

In the previous chapters, we discussed how researchers extract and prepare software engineering process data and introduced an extension of the original linking approach, achieving a better linking ratio than previously presented. Like other researchers, we make several assumptions about the way how bugs are reported, fixed, and linked. We then presented a framework of data quality and characteristics measures and used this framework to evaluate all projects investigated in this thesis. We found that all datasets are plagued by quality issues and that we have vast differences in characteristics of data across all projects.

But why should researchers care about these issues? In this part we try to answer this question by analyzing three hypotheses:

HYPOTHESIS 3.1: *Process assumptions made by empirical software engineering researchers may be wrong.*

HYPOTHESIS 3.2: *Software engineering datasets are plagued by bias due to a lack of complete linking.*

[1]Parts of this chapter have already been published [Bachmann et al., 2010]

HYPOTHESIS 3.3: *Bias in software engineering datasets has an effect on empirical software engineering results.*

In this chapter we focus on Hypothesis 3.1. We try here to verify process assumptions made by researchers. They assume, for instance, that all the relevant bugs of a software product are actually reported in the BTS of the project. To truly understand defect-reporting practices and verify such assumptions, we must uncover the *ground truth*: We must analyze completely (at least a time-window of) the commit version history of a project, and precisely identify all the commits that are defect fixes, and those that are not.

To get at the ground truth requires skill, knowledge, and effort: One must compare successive versions, understand the changes, identify any relevant reported bugs in the VCS, and establish a link to the BTS when possible. This process must be repeated until we have a large enough sample for statistical analysis. Unfortunately, this currently requires an expert to manually extract information from multiple sources and analyze it.

Specifically, we engaged an expert APACHE core developer, Dr. Justin Erenkrantz, to manually annotate the six-week period of our APACHE evaluation sample dataset. Based on the results and interviews, we are able to present a detailed view in the practices of a very popular and often-used OSS project. In addition, we show that software engineering datasets are more plagued by quality issues as thought before and process assumptions made by empirical software engineering researchers may be wrong, such as the most relevant bugs of APACHE never showing up in the APACHE BTS, instead being discussed on the APACHE email discussion system.

8.1 APACHE Evaluation Procedure

To address Hypothesis 3.1 and get to the ground truth in our APACHE evaluation sample dataset, we have to analyze all the commits and identify those that are bug fixes and those that are not. Indeed, an-

notating program code commits done months to years in the past is a challenge, even for a core developer like Justin. Therefore, to address the difficulties of performing annotations we used a tool called LINKSTER, which was developed by Christian Bird at the University of California, Davis [Bachmann et al., 2010; Bird et al., 2010].

LINKSTER is a convenient, interactive tool, integrating multiple queryable, browseable, time-series views of VCS and BTS history. LINKSTER enables an expert to quickly find and examine relevant changes, and annotate them as desired; specifically, LINKSTER makes it easy to find defect-fix commits.

Using LINKSTER, Justin annotated each commit, flagging it as a *bug fix*, an implemented *feature request*, a *maintenance* task, or *other*. With this information, we obtained fully annotated commit data, providing a complete picture of all the changes during the given period and how, why, and by whom these changes were made. This data can be used to verify our automated linking approach (which includes mainly bug fixes and some feature requests). LINKSTER provides an integrated view of all the relevant information to annotate commits. Based on the log message, the changed files and the file diffs of the changed files, Justin was able to annotate all commits, and, in most cases, provided additional information about the commits.

Justin's familiarity with the APACHE project gives us confidence that the results of our evaluation can be trusted. In addition, detailed discussions and interviews with him revealed facts about the tools and processes used in the APACHE HTTP WEB SERVER project. In the next section we discuss the results of our APACHE evaluation.

8.2 APACHE Evaluation Results

Using LINKSTER, Justin was able to annotate all 493 commits in our APACHE evaluation sample dataset (see Section 4.1.1). In addition to the annotation into the four categories above—*bug fix*, *feature request*, *maintenance/ refactoring*, and *other*—our informant helped us further sub-classify the commits. Table 8.1 summarizes the annotation results

including the sub-classification. Note that a single commit can have many annotations; for example, a commit may be annotated as both a "bug fix" and a "feature request".

Table 8.1: Commit categorization of APACHE $_{evaluation}$ (non-exclusive)

Category	Sub-category	Number of commits
Bug fix	–	82
Bug fix	Bug report	32
Bug fix	Bug report (merge)	7
Bug fix	Email discussion system	13
Bug fix	Backport	13
Bug fix	Other	17
Feature request	–	54
Feature request	Documentation	7
Feature request	Backport	14
Feature request	Other	33
Maintenance	–	49
Maintenance	Documentation	5
Maintenance	Backport	5
Maintenance	Other	39
Other	–	356
Other	Documentation	156
Other	Backport	49
Other	Non-functional	30
Other	Release	44
Other	Voting	26
Other	Other	51

Based on Justin's insights into the APACHE development process, we developed a second, orthogonal categorization that was more consistent with the procedures within the project (Table 8.2). In contrast to our categorization, this one assigns each commit exclusively to one of its process-specific categories: *backport/forward port*, *security fix*, *bug fix*, *documentation*, *voting*, *release*, or *other*.

Table 8.2: Process-specific commit categorization of APACHE $_{evaluation}$ (exclusive)

Category	Number of commits
Backport / forward port	79
Security fix	7
Bug fix	69
Documentation	158
Voting	26
Release	44
Other	110

In the following sub-sections, we present our findings relative to Hypothesis 3.1. We also present additional findings based on interviews with Justin.

8.2.1 Bugs Incognito

The APACHE project makes use of BUGZILLA as BTS and SVN as VCS (see Table 4.1). In addition, APACHE maintains a publicly-readable email discussion system on which developers and APACHE users are able to discuss project issues, future releases, bugs, and exchange ideas about the configuration of the APACHE HTTP WEB SERVER. Contrary to conventional wisdom, participants in the APACHE project *do not report all the bugs solely through the BTS*. We found that developers and professional users also make use of the APACHE email discussion system to report bugs and provide bug fixes (sometimes at the same time) without reporting them in the bug tracker.

FINDING 8.1. *Not all fixed bugs are mentioned in the BTS. Some are discussed (only) on the email discussion system.*

As shown in Table 8.1, we have 82 bug fix related commits in our evaluation dataset. 32 of them (bug report) are directly related to the BTS. Seven other commits contain a bug fix, but are not the initial bug

fix commit, but rather a merge of versions which contain bug fixes indirectly (bug report (merge)). This means that only 47.6% of bug fix related commits ($\frac{32+7}{82}$) are documented in the BTS. For 13 other commits (16% of total) identified by Justin as bug fixes, there are related discussions in the APACHE email discussion system.

This leads to the discouraging observation that many bugs never appear in the BTS, but rather are *only* discussed on the email discussion system. Such a discussion often includes the bug fix provided by someone other than an APACHE core developer. According to Justin, these bugs are often the very important bugs, especially because of the high attention by APACHE developers and the core community on the email discussion system. Note also that reporting some types of bugs (e.g., security related ones) on the email discussion system is a practice explicitly requested by the APACHE Foundation[2].

Unfortunately, even knowing about the email discussion system bugs, it is hard to (i) identify and (ii) automatically mine them or extract information similar to a bug report stored in the BTS (such as status changes, priority, severity, etc.). APACHE SVN revision #291558 (see Figure 8.1), for instance, is related to a bug discussed on the email discussion system (Figure 8.2). If one were to inspect the email discussion system message, one would find almost no evidence that this was a bug fix.

```
------------------------------------------------------------------------
r291558 | pquerna | 2005-09-26 06:44:16 +0200 (Mon, 26 Sep 2005) | 2 lines
Changed paths:
   M /httpd/httpd/trunk/Makefile.in

As recommended by nd, build docs for all languages.

------------------------------------------------------------------------
```

Figure 8.1: Commit message of APACHE HTTP WEB SERVER revision #291558

[2]See `http://httpd.apache.org/security_report.html`

Mailing list archives

Site index · List index

Message view	« Date » · « Thread »
From	André Malo <..@perlig.de>
Subject	Re: svn commit: r291480 - in /httpd/httpd/trunk: Makefile.in build/rules.mk.in
Date	Mon, 26 Sep 2005 04:27:33 GMT

```
* pquerna@apache.org wrote:

> +docs:
> +       @if test -d $(top_srcdir)/docs/manual/build; then \
> +           cd $(top_srcdir)/docs/manual/build && ./build.sh; \
> +       else \
> +           echo 'For details on generating the docs, please read:'; \
> +           echo '  http://httpd.apache.org/docs-project/docsformat.html'; \
> +       fi

FWIW, ./build.sh builds only the English files. You might want to use
`./build.sh all`.

nd
--
```

Figure 8.2: Email discussion (extract) for APACHE HTTP WEB SERVER revision #291558[3]

Finally, Justin found 17 other bug fixing commits (21%) which have neither an associated bug report nor email discussion system message. This phenomenon of the under-reporting of bugs is a big problem: Empirical software engineering research relies on data from BTSs, VCSs, and the linkage between these two data sources. If important bugs are excluded from experimental data (i.e., many bugs are left out), then the effectiveness of defect prediction models and the validity of statistical studies (which rely on them being in the BTS) may be threatened. This leads to the conclusion that not all fixed bugs are reported as bugs in the BTS, or in other words bugs go "incognito". This contradicts assumptions commonly made by researchers and, unfortunately, supports Hypothesis 3.1.

8.2.2 Backport Incognito

In the APACHE HTTP WEB SERVER project only a few developers are allowed to commit to an APACHE release version; thus a bug fix on one release therefore have to be committed by someone else to an older or different release.

[3]See http://mail-archives.apache.org/mod_mbox/httpd-docs/200509.mbox/%3c200509260627.33737@news.perlig.de%3e

Typically, this process works as follows:

1. A developer fixes a given bug and commits the new version to the current version under active development (also known as the "trunk"). Ideally s/he also refers to the related bug report in the commit log.

2. At least two other developers review the changed code, verify the changes, and vote either for or against the fix (this step is related to the voting commits as shown in Tables 8.1 and 8.2).

3. If the vote is positive, the fix is committed (or merged) to APACHE release versions, which is called a backport.

As a result of this process, we might find several different commits in the version history, that fix the same bug.

FINDING 8.2. *To fix a bug in an* APACHE *release, multiple similar commits by different developers are needed.*

This process raises an interesting question: Which commits are counted as bug fixes for a reported bug? There are reasons why both commits—the original commit to trunk and the backport commit(s)—should be counted as the bug fix. Nevertheless, we believe that mainly the backport commit should count as the bug fix because only this commit really fixes the bug in a given release of the software product. We can compare the commit to trunk with a commit to a testing and quality assurance environment and the backport commit(s) as commit(s) to the productive environment. The distinction between trunk and release is simply the different SVN paths.

Unfortunately, backport commits are not that easy to identify by existing linking algorithms and heuristics; frequently, while the log message for original commit to the trunk refers to the bug report, the backport commit log does not. To worsen matters, after the bug is actually closed, there is a rigorous review, verification, and voting process before the backport is accepted and committed. Therefore, the time difference between the backport commit and the status change

(to fixed) on the bug report may increase to several days, which again makes it difficult to link the bug with the commit. As a result, automated linking algorithms will largely ignore backport fixes. Arguably, these fixes are very important; they are often involved in post-release failures. They should not be ignored by researchers engaged in hypothesis testing or defect prediction work. Alas, finding them may require extensive, high-expertise combing through commit histories.

8.2.3 Impact-of-Defect vs. Cause-of-Defect

This is a thorny issue: A defect in one project's code base might actually manifest as a failure in a different project. Thus, some of the reported bugs in APACHE HTTP WEB SERVER have their root cause outside of the APACHE program code. APACHE uses external libraries as well as Apache Commons modules. Therefore, failures in the APACHE HTTP WEB SERVER, even if duly reported in the APACHE BTS, may actually have to be fixed elsewhere. The reverse is also possible.

The mod-python[4] sub-project maintains its own VCS repository and an APACHE project's main BTS independent Jira issue tracker[5]. Mod-python issue 83[6], for instance, was reported in the Jira issue tracker but fixed in the APACHE program code.

FINDING 8.3. *Developers sometimes fix bugs that are only reported in some other projects' BTS, rather than in their own, and vice-versa.*

Ideally, we have a complete, integrated source of all the bugs in the BTS, and all the fixes in the VCS. Based on our findings, and indeed the widespread prevalence of cross-project module reuse, we can expect that this type of separation between causes and effects of defects is quite common. Given this, it would be helpful if a report of a bug *impacting* one system would be transferred to the BTS of the *causing* system, and linked to fix in the VCS of that system. However, given

[4] `http://www.modpython.org/`
[5] `http://www.atlassian.com/software/jira/` and `https://issues.apache.org/jira/browse/MODPYTHON`
[6] `https://issues.apache.org/jira/browse/MODPYTHON-83`

the poor linking behavior when the cause and effect are in the *same* system, we might expect that this type of cross-system linking is pretty unlikely to occur.

Indeed, this phenomena is not a problem *per se* if we ensure that the program code which is stored in the VCS uses the BTS we have data for. Luckily, this is often the case. If not, we may have another source of bugs that may be incognito.

8.2.4 Commits Incognito

In the previous chapter, we encountered the problem of unexplained commits, for example, due to empty commit log messages. Sadly, even an experienced developer will find it difficult to retrospectively reconstruct the explanation of an unexplained commit.

FINDING 8.4. *Even if we annotate all commits, the cause of a commit still remains unspecified in some cases.*

Table 8.1 and 8.2 show the annotation, sub-classification, and process-oriented classification of all the commits in our APACHE evaluation sample dataset. Based on the values in Table 8.2, for 110 commits (22.3%) we have a process-specific annotation of other. The reason for these commits, therefore, is not justified by one of the APACHE software engineering core tasks.

In addition, most of the commits are not justified by a bug fix or feature request , but instead for documentation (32%), voting (5.3%), or releases (8.9%). Only 37.1% of all commits have a functional impact on the software product (feature requests and bug fixes including all backports), which leads us to the conclusion that not all commits are commits that actually change the software.

8.3 Threats to Validity

Generalizability of Results. Can we generalize from the results based on the APACHE HTTP WEB SERVER dataset to other datasets? Software engineering tools and processes vary in different projects (as discussed in the previous chapter) and, therefore, our findings based on APACHE may not generalize. However, our findings indicate that developers may use software process support tools for various goals not envisioned by its original developers (such as VCSs for voting or email discussion systems for bug reporting). It seems prudent to assume that the APACHE project is not a complete exception and that, therefore, the data used in studies of other projects may also lack important information. Another threat is the use of a single annotator (Justin). Getting the same data annotated by other developers, and checking agreement, would have been better.

Evaluation Sample Dataset. Did we choose our APACHE evaluation sample dataset well, and properly analyze it? We chose our timeframe carefully; however, it may not properly represent the original APACHE dataset.

Annotation Validity. The annotation and classification were performed carefully by a very experienced APACHE core developer and based on his familiarity with the APACHE project, we are highly confident that the results of our evaluation are trustworthy. Still, there may be errors. Nonetheless, according to Justin, the interesting practices of the APACHE developers are by no means exceptional to this time period.

8.4 Concluding Discussion

In this chapter, we tried to find the "ground truth" in the commit annotations of a very popular empirical software engineering dataset. We used temporal sampling to define an evaluation subset of the original APACHE dataset and manually annotated all commits, with the assistance of an APACHE core developer and the use of LINKSTER.

In the previous section, we showed that software engineering datasets are plagued by quality issues. Unfortunately, based on our APACHE data evaluation, we found that things are even worse: Our findings cast doubt on some of the core assumptions made in empirical research. Specifically:

1. Bugs often go incognito as they are not always reported as a bug in the BTS but, for example, in email discussion systems,
2. commits do not always clearly change the functionality of the program, and
3. bugs and commits often point to changes documented in other projects and are, therefore, badly captured by the project tools.

Specifically, we showed that not all fixed bugs are reported in the BTS and most of the commits (62.9%) are not related to a bug fix or feature request (which would introduce a program change), but instead for documentation (32%), voting (5.3%), or releases (8.9%). In addition, we presented the curious case of backport commits and the challenging impact-of-defect vs. cause-of-defect problem. Both issues have an impact on software engineering datasets.

Consequently, even though automated linkage tools are able to connect a remarkable number of commits to bug reports, many bugs—sometimes the most critical ones—never show up in the BTS and are, therefore, not linked. This raises new issues concerning the validity of studies that rely on VCS logs and BTS data only and, unfortunately, supports Hypothesis 3.1.

Another implication of our findings is that empirical software engineering studies will need to take the whole software development social eco-system (revision control system, bug tracking database, mailing list systems, email discussions, discussion boards, chats, etc. as well as these data from other, related projects) into account in order to elicit a more complete picture of the underlying development process. This would allow the capturing of the nature of some of the bugs and commits that our informant tediously collected manually.

Summarizing, we have shown that there are even more serious and consequential problems in software engineering datasets than previ-

ously suspected. Since only a sample of fixed bugs are mentioned and, therefore, linked to the VCS, in the next chapters we go a step further and (i) check all datasets for bias and (ii) analyze possible threats to validity in research results by these issues. Specifically, we analyze Hypotheses 3.2 and 3.3.

9

Bias in Software Engineering Datasets[1]

In recent years, empirical software engineering researchers have focused much effort on two critical areas. First, understanding the causes of poor software quality, and second, on building effective bug prediction systems. Researchers taking the first approach formulate hypotheses of defect introduction (more complex code is more error-prone code, pair-programming reduces defects, etc.) and then use field data concerning defect occurrence to statistically test these theories. Researchers in bug prediction systems have used historical bug fixing field data from software projects to build prediction models (e.g., based on machine learning). Both areas are of enormous practical importance: The first can lead to better practices, and the second to more effectively targeted inspection and testing efforts. Both approaches, however, strongly depend on good data sets to find the precise location of bug introduction. In the previous sections, we have shown that such datasets are plagued by quality issues such that only a sample of the fixed bug reports are mentioned in commit log messages and, therefore, are recognized by linking algorithms.

[1]Major parts of this chapter have already been published [Bachmann et al., 2010; Bird et al., 2009a]

However, predictions made from samples can be wrong if the samples are not representative of the population. The effects of bias in survey data, for instance, are well-known [Easterbrook et al., 2007]. If there is some systematic relationship between the choice or ability of a surveyed individual to respond and the characteristic or process being studied, then the results of the survey may not be accurate. A classic example of this are the predictions made from political surveys conducted via telephone in the 1948 United States presidential election [Levy, 1983]. At that time, telephones were mostly owned by individuals who were wealthy, which had a direct relationship with their political affiliation. Thus, the (incorrectly) predicted outcome of the election was based on data in which one political party was overrepresented. Although telephone recipients were chosen at random for the sample, the fact that a respondent needed to own a telephone to participate introduced sampling bias into the data.

Sampling bias is a form of non-response bias because data is missing from the full population [Singleton and Straits, 2009], making a truly random sample impossible. Likewise, bug fix data sets, for example, might over-represent bug fixes performed by more experienced or core developers, perhaps because they are more careful about reporting. Hypothesis testing on this dataset might lead to the incorrect conclusion that more experienced developers make more mistakes; in addition, prediction models might tend to incorrectly signal a greater likelihood of error in code written by experienced developers, and neglect defects elsewhere.

In this section, therefore, we analyze the following hypotheses:

HYPOTHESIS 3.2: *Software engineering datasets are plagued by bias due to a lack of complete linking.*

HYPOTHESIS 3.3: *Bias in software engineering datasets has an effect on empirical software engineering results.*

To do so, we characterize the bias problem in software engineering datasets from two different perspectives: Bug feature bias, where only certain types of bugs are linked, and commit feature bias, where

only the certain kinds of fixes, or fixes to certain kinds of files, are linked. We also discuss the consequences of each type of bias. We then quantitatively evaluate our datasets for bug feature bias, analyze several sub-hypotheses and evaluate the performance of BUGCACHE, an award-winning bug prediction system, on bug feature biased data to analyze the potential effect of bias on research results (Hypothesis 3.3). The analysis of datasets on commit feature bias, in contrast, needs more attention since we usually do not have a fully annotated set of commits that tells us whose commits are bug fixes and whose are not. Therefore, we make use of the fully annotated APACHE evaluation sample we introduced in the previous chapter. This fully annotated dataset enables the analysis of the data for commit feature bias. Nonetheless, we remind the reader that our APACHE evaluation sample size is not big enough to realistically expect to find statistically significant support for hypothesis testing. In Sections 9.4 and 9.5 we therefore analyze a set of questions related to Hypotheses 3.2 and 3.3.

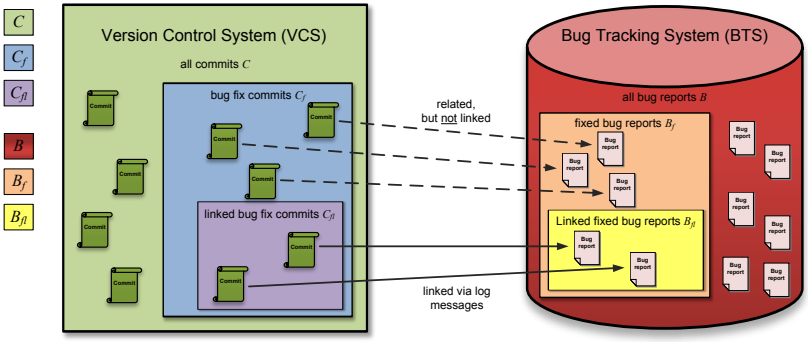


Figure 9.1: Sources of bug data and commit data and their relationships

9.1 Background and Theory

Figure 9.1 illustrates the various sources of software engineering data that have been used for both hypothesis testing and bug prediction in the past. Many projects use BTSs, with information about reported bugs (right of figure). We denote the entire set of bugs in the BTS as B. Some of these reported bug have been fixed by making changes to the program code, and been marked fixed; we denote these as B_f. On the left we show the VCS, containing every revision of every file. This records (for every revision) who made the commit, the time, the content of the change, and the log message. We denote the full set of commits as C. The subset of C, which represents commits to fix bugs reported in the bug dataset, is C_f. Unfortunately, there is not always a link between the fixed bugs in the BTS and the VCS commits that contains those fixes. In general, therefore, C_f is only partially known.

To infer the relationship between the commits C_f and the fixed bugs B_f one can use commit log information to establish links based on heuristics. In Section 5.4 we introduced an extended linking algorithm to do so. However, these techniques depend on developers recording identifiers, such as bug report numbers in commit log messages. Likewise, only a portion of the fixed bugs in the database can be tied to their corresponding program code changes. We denote this set of "linked" bug fix commits as C_{fl} and the set of linked bugs in the bug repository as B_{fl}.

Unfortunately, $|B_{fl}|$ is usually quite a bit smaller than $|B_f|$, as discussed in Section 7.2. Consequently, there are many bug fixes in C_f that are not in C_{fl}. Therefore we also conclude that $|C_{fl}| < |C_f|$. The critical issue is this: Programmers fix bugs, but they only sometimes explicitly indicate (in the commit logs) which commits fix which bugs. While we can thus identify C_{fl} by pattern-matching (e.g., as discussed in Chapter 5), identifying C_f requires extensive retro perspective manual effort, and is usually infeasible.

9.1.1 Features

Both experimental hypothesis testing and bug prediction systems make use of measurable features. Prediction models are usually cast as a classification problem; given a fresh commit $c \in C$, classify it as "good" or "bad". Often, predictors use a set of *commit features* (such as size, complexity, or churn) $f_1^c \ldots f_m^c$, each with values drawn from domains $D_1^c \ldots D_m^c$, and perform the classification operation using a prediction function F_p:

$$F_p : D_1^c \times \ldots \times D_m^c \rightarrow \{Good, Bad\} \tag{9.1}$$

Meanwhile, bugs in the bug database also have their own set of features, which we call *bug features*, which capture the properties of the bug and its history. Properties include severity of the bug, the number of people working on it, how long it remains open, the experience of the person finally closing the bug, and so on. By analogy with commit features, we define *bug features* $f_1^b \ldots f_n^b$, with values drawn from domains $D_1^b \ldots D_n^b$. We note here that both commit features and bug features can be measured for the entire set of bugs and the entire set of commits. However, B_{fl} represents only a portion of the fixed bugs, B_f. Similarly, we can only examine C_{fl}, since the full set of bug fixing commits, C_f, is usually unknown. The question that we pose is: Are the sets B_{fl} and C_{fl} representative of B_f and C_f respectively, or is there some sort of bias? Next, we more formally define this notion, first for bug features, and then for commit features.

9.1.2 Bug Feature Bias

Consider the set B_{fl}, representing bugs whose repair is linked to source files. Ideally, all types fixed bugs would be equally well-represented in this set. If this were the case, predictive models and hypotheses of defect causation would use data concerning every type of bug. If not, it is possible that certain types of bugs might be systematically omitted from B_{fl}, and thus any specific phenomena pertaining to these bugs would not be considered in the predictive models and/or hypothesis

testing. Informally, we would like to believe that the properties of the bugs in B_{fl} look just like the properties of all fixed bugs. Stated in terms of conditional probability, the distributions of the bug features over the linked bugs and all fixed bugs would be equal:

$$p(f_1^b \dots f_n^b \mid B_{fl}) = p(f_1^b \dots f_n^b \mid B_f) \tag{9.2}$$

If Eq 9.2 above is not true, then bugs with certain properties could be over- or under-represented among the linked bugs; this might lead to poor bug prediction and/or threaten the external validity of hypothesis testing. We call this *bug feature bias*.

9.1.3 Commit Feature Bias

Commit features can be used in a *predictive* mode, or for hypothesis testing. Given a commit c that changes or deletes code, we can use version history to identify the prior commits that introduced the code that c affected. The affected code might have been introduced in more than one commit. Most VCSs include a "blame" command which, given a commit c, returns a set of commits that originally introduced the code modified by c:

$$blame : C \longrightarrow 2^C \tag{9.3}$$

Without ambiguity, we can promote $blame$ to work with sets of commits as well; thus, given the set of linked commits C_{fl}, we can meaningfully refer to $blame(C_{fl})$ the set of commits that introduced code that were later repaired by linked commits, as well as $blame(C_f)$, the set of all commits that contained code that were later subject to defect repair. Ideally, there is nothing special about the linked, blame set $blame(C_{fl})$, as far as commit features are concerned:

$$p(f_1^c \dots f_m^c \mid blame(C_{fl})) = p(f_1^c \dots f_m^c \mid blame(C_f)) \tag{9.4}$$

If Eq 9.4 does not hold, that suggests that certain types of commit features (such as size, complexity, authorship, etc.) are being systematically over-represented (or under-represented) among the linked

bugs. We call this *commit feature bias*. This would bode ill both for the accuracy of prediction models and for the external validity of hypothesis testing that made use of the features of the linked blame set $blame(C_{fl})$.

The empirical distribution of the commit features properties on the linked blame set, $blame(C_{fl})$, can certainly be determined. The real problem here, again, is that we have no (automated) way of identifying the exact set of bug fixes, C_f. Therefore, in general, we come to the following rather disappointing conclusion: Given a set of linked commits C_{fl}, there is no way of knowing if commit feature bias exists, due to a lack of access to the full set of bug fix commits C_f. Consequently, we are able to analyze commit feature bias in our APACHE evaluation sample dataset only.

9.2 Analysis of Bug Feature Bias

In this section we examine several possible types of bug feature bias in B_{fl}. We consider features relating to three general categories: the bug type, properties of the bug fixer, and properties of the fixing process. The combined results of all our tests are shown in Table 9.1. We note here that all p-values have been corrected for multiple hypothesis testing, using the Benjamini-Hochberg correction [Benjamini and Hochberg, 1995]; the correction also accounted for the hypotheses that were not supported. In addition to p-values, we also report summary statistics (rows 4/5 and 7/8) indicating the magnitude of the difference between observed feature values in linked and unlinked samples. With large sample sizes, even small-magnitude, unimportant differences can lead to very low p-values. We therefore also report summary statistics, so that the reader can judge the significance of the differences.

Table 9.1: Bug feature bias hypothesis testing results for each of the projects

		APACHE	ECLIPSE	GNOME	NETBEANS
1	Total fixed bugs	1299	101551	102422	N/A
2	Linked fixed bugs	615	35266	42357	N/A
3	Severity χ^2	$p \ll .01$	$p \ll .01$	$p \ll .01$	N/A
4	Med. exp. for all	25	178	175	213
5	Med. exp. for linked	31	408	213	258
6	Experience KS	$p \ll .01$	$p \ll .01$	$p = .08$	$p \ll .01$
7	Verified $\hat{\pi}$ for all	.006	.317	.016	.631
8	Verified $\hat{\pi}$ for linked	.006	.492	.013	.694
9	Verified χ^2	$p = .99$	$p \ll .01$	$p = .99$	$p \ll .01$

		OPENOFFICE	MOZILLA	BSZKB#1	BSZKB#2
1	Total fixed bugs	N/A	150424	5739	305
2	Linked fixed bugs	N/A	52820	1657	109
3	Severity χ^2	N/A	$p \ll .01$	$p \ll .01$	$p \ll .01$
4	Med. exp. for all	116	160	N/A	N/A
5	Med. exp. for linked	167	238	N/A	N/A
6	Experience KS	$p \ll .01$	$p \ll .01$	N/A	N/A
7	Verified $\hat{\pi}$ for all	.650	N/A	N/A	N/A
8	Verified $\hat{\pi}$ for linked	.881	N/A	N/A	N/A
9	Verified χ^2	$p \ll .01$	N/A	N/A	N/A

9.2.1 Bug Type Feature: Severity

In the BTSs of all projects investigated except for OPENOFFICE and NETBEANS, bug reports are given a severity level. This ranges from blocker—defined as "Prevents function from being used, no work around, blocking progress on multiple fronts"—to trivial—"A problem not affecting the actual function, a typo would be an example".[2] The CSS projects use a similar but simplified range. Certainly bug severity is important as developers are probably more concerned with more severe bug reports which inhibit functionality and use. Given the importance of more severe bug reports, one might reasonably assume that the more severe bug reports are handled with greater care, and therefore are more likely to be linked. We therefore believe that bug reports in the more severe categories are over-represented in B_{fl}, and that we observe bug feature bias based on the severity of the bug.

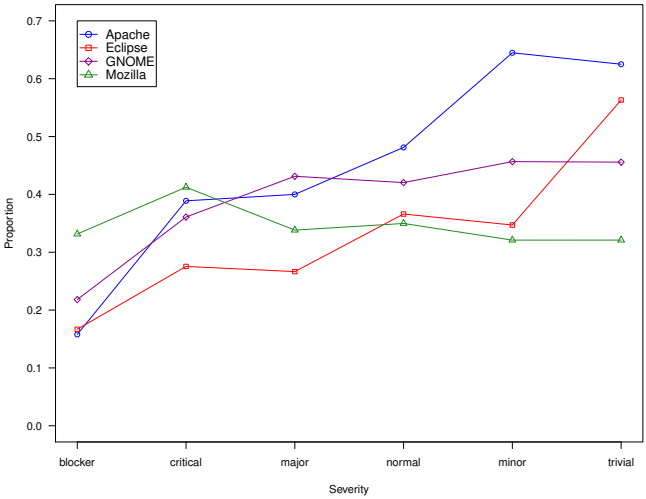
HYPOTHESIS 3.2.1: *There is a difference in the distribution of severity levels between B_{fl} and B_f.*

Figure 9.2 shows the proportion of fixed bug reports that can be linked to specific commits, broken down by severity level. In the APACHE project, 63% of the fixed minor bugs are linked, but only 15% of the fixed blocker bugs. Quite similar results we found for ECLIPSE. If one were to undertake hypothesis testing or train a prediction model on the linked fixed bugs, the minor and trivial bugs would be over-represented. We seek to test if:

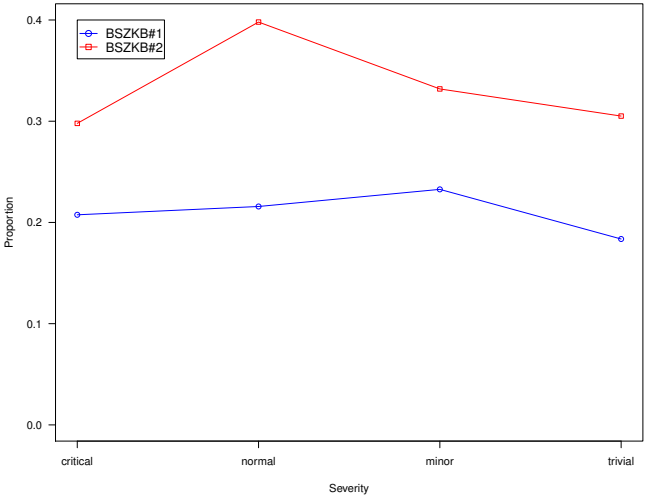
$$p(severity \mid B_{fl}) = p(severity \mid B_f) \tag{9.5}$$

Note that severity is a categorical (and in some ways ordinal) value. It is clear in this case that there is a difference in the proportion of linked bugs by category. For a statistical test of the above equation we use Pearson's χ^2 test [Dowdy et al., 2004] to quantitatively evaluate if the distribution of severity levels in B_{fl} is representative of B_f.

[2] See `http://www.eclipse.org/tptp/home/documents/process/development/bugzilla.html`



(a)



(b)

Figure 9.2: Proportion of fixed bug reports that are linked (by severity level)

With five severity levels, the observed data yields a χ^2 statistic value of 94 for APACHE (with five degrees of freedom), and vanishingly low p-values in the case of APACHE, ECLIPSE, GNOME, MOZILLA, BSZKB#1, and BSZKB#2 (row 3 in Table 9.1). This indicates that it is extremely unlikely that we would observe this distribution of severity levels if the bugs in B_f and B_{fl} were drawn from the same distribution. We were unable to perform this study on the OPENOFFICE and NETBEANS data sets because their BTSs do not include a severity field on all bugs. Surprisingly, in each of the OSS projects we observed a similar trend: The proportion of fixed bugs that were linked decreased as the severity increased. Interestingly, the CSS projects do not have such strong tendencies and, therefore, seem to be less affected by this type of bug feature bias. The p-values are, indeed, on a significant level and therefore Hypothesis 3.2.1 is supported for all projects for which we have severity data.

Our data also indicates that B_{fl} is biased towards less severe bug categories. A defect prediction technique that uses B_{fl} as an oracle will actually be trained with a higher weight on less severe bugs. If there is a relationship between the features used in the model and the severity in the bug (e.g., if most bugs in the GUI are considered minor), then the prediction model will be biased with respect to B_f and will not perform as well on more severe bugs. This is likely the exact opposite of what users of the model would like. Likewise, testing hypotheses concerning mechanisms of bug introduction on these datasets, might lead to conclusions more applicable to the less important bugs.

9.2.2 Bug Fixer Feature: Experience

We theorize that more experienced project members are more likely to explicitly link bug fixes to their corresponding commits. The intuition behind this hypothesis is that project members gain process discipline with experience, and that those who do not, over time, are replaced by those who do.

HYPOTHESIS 3.2.2: *Bug reports in B_{fl} are fixed by more experienced people than those who fix bugs in B_f.*

Here, we use the experience of the person who marked the bug record as fixed. We define the experience of a person at time t as the number of bug records that person has marked as fixed prior to t. Using this measure we record the experience of the person marking the bug as fixed at the fixing time. In this case we will test if:

$$p(experience \mid B_{fl}) = p(experience \mid B_f) \tag{9.6}$$

Experience in this context is a continuous variable. Therefore, we use a Kolmogorov-Smirnov test [Conover, 1998], a non-parametric, two-sample test indicating if the samples are drawn from the same continuous distribution. Since we hypothesize that the experience for linked bugs is higher (rather than just different), we use a one-sided Kolmogorov-Smirnov test.

To illustrate this, Figure 9.3 shows boxplots of experience of bug closers for all fixed bugs and for the linked bugs. The Kolmogorov-Smirnov test finds a statistically significant difference (although somewhat weak for APACHE), in the same direction between B_{fl} and B_f in every case. Rows 4 and 5 of Table 9.1 show the median experience for closers of linked bugs and all fixed bugs. The distribution of experience is heavily skewed, so the median is in this case a better summary statistic than the mean. Hypothesis 3.2.2 is therefore confirmed for all datasets tested. Unfortunately, we were unable to test this hypothesis for the ZKB datasets due to lack of information in the BTSs.

The implications of this result are that more experienced bug closers are over-represented in B_{fl}. As a consequence, any defect prediction model is more likely to predict bugs marked fixed (though perhaps not committed) by experienced developers; hypothesis testing on these data sets might also tend to emphasize bugs of greater interest to experienced bug closers.

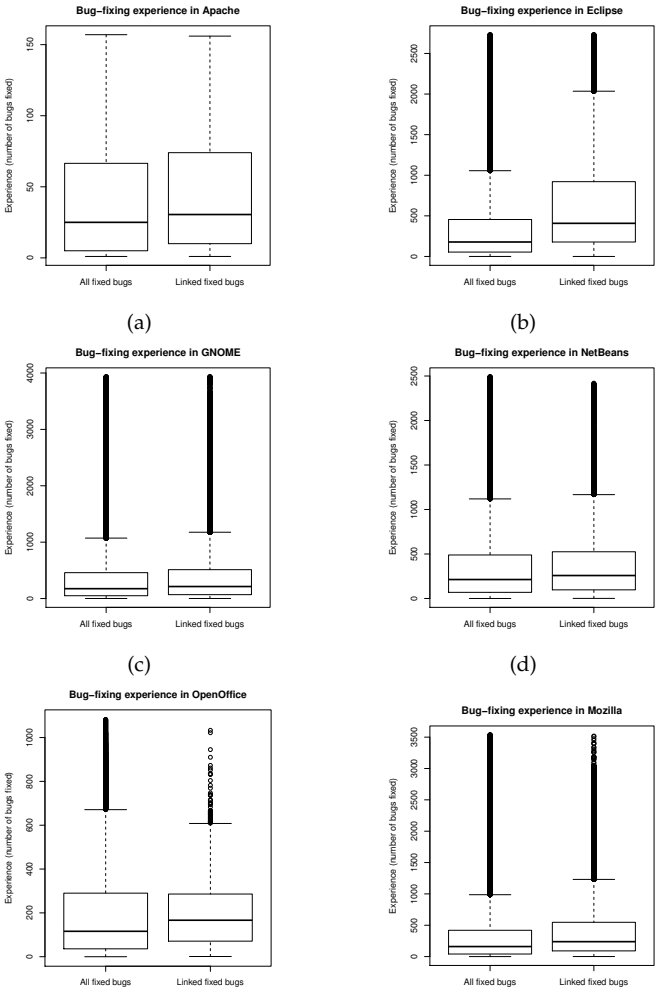


Figure 9.3: Boxplots of experience of bug closer for all fixed bug records and linked bug records

9.2.3 Bug Process Feature: Verification

The BTS for each of the projects investigated records information about the process that a bug report goes through. Once a bug has been marked as resolved, it may be verified as having been fixed, or it may be closed without verification (see Section 3.1.3 and [Gasser and Ripoche, 2003]). We hypothesize that being verified indicates that a bug is important and will be related to linking.

HYPOTHESIS 3.2.3: *Bugs in B_{fl} are more likely to have been verified than the population of fixed bugs, B_f.*

We test if:

$$p(\textit{verified} \mid B_{fl}) = p(\textit{verified} \mid B_f) \tag{9.7}$$

While it is possible for a bug to be verified more than once, we observe that this is rare in practice. Thus, the feature *verified* is a *dichotomous* variable. Since the sample size is large, we can again use a χ^2 test to determine if verification is different for B_{fl} and B_f.

In addition, since *verified* is a binomial variable, we can compute the 95% confidence interval for the binomial probability $\hat{\pi}$ of a bug being verified in B_{fl} and B_f. We do this using the Wilson score interval method [Agresti and Coull, 1998]. For the ECLIPSE data, the confidence interval for linked bugs is $(0.485, 0.495)$, with point estimate 0.490. The confidence interval for the population of all fixed bugs is $(0.289, 0.293)$, with point estimate 0.291. This indicates that a bug is 66% more likely to be verified if it is linked. No causal relationship has been established; however, the probability of a bug being verified is conditionally dependent on it also being linked. Likewise, via Bayes' Theorem [Mitchell, 1997], the probability of a bug being linked is conditionally dependent on it being verified.

The Pearson χ^2 results and the binomial probability estimates $\hat{\pi}$ are shown in rows 7–9 of Table 9.1. In all cases, the size of the confidence interval for $\hat{\pi}$ was less than .015. We found that there was bias with respect to being verified in ECLIPSE, NETBEANS, and OPENOFFICE. There was no bias in APACHE. Interestingly, in GNOME, bugs

that were linked were less likely to have been verified. We therefore confirm Hypothesis 3.2.3 for ECLIPSE, NETBEANS, and OPENOFFICE.

9.2.4 Bug Process Features: Miscellaneous

We also evaluated a number of hypotheses, relating process-related bug features, that were *not* confirmed.

HYPOTHESIS 3.2.4: *Bugs that are linked are more likely to have been closed later in the project than bugs that are not.*

The intuition behind this is that the policies and development practices within projects tend to become more rigorous as the project grows older and more mature. Thus we conjecture that the proportion of fixed bugs that are linked will increase with the lifetime of the project. However, this hypothesis was not confirmed in the projects studied.

HYPOTHESIS 3.2.5: *Bugs that are closed near a project release date are less likely to be linked.*

As a project nears a release date (for projects that attempt to adhere to release dates), the BTS becomes more and more important as the ability to release is usually dependent on certain bugs being fixed. We expect that with more time pressure and more (and possibly less experienced) people fixing bugs at a rapid rate, fewer bugs will be linked. We found no evidence of this in the data.

HYPOTHESIS 3.2.6: *Bugs that are linked will have more people or events associated with them than bugs that are not.*

Bug records for the projects studied track the number of people that contributed information to them in some way (comments, assignments, triage notations, etc.). They also include the number of "events" (e.g., severity changes, reassignments) that happen in the life of the bug. We hypothesize that more people and events would be

associated with more important bugs, and thus contribute to a higher likelihood of linkage. The data did not support this hypothesis.

9.3 Effects of Bug Feature Bias

We now turn to the critical question: *Does bug feature bias matter?* Bias, in a sample, matters only insofar as it affects the hypothesis that one is testing with the sample, or the performance of the prediction model trained on the sample. We now describe an evaluation of the impact of bug feature bias on a defect prediction model, specifically, BUG-CACHE, an award-winning method by Kim *et al.* [Kim et al., 2007].

If we train a predictor on a biased training set, which is biased with respect to some bug features, how will that predictor perform on the unbiased full population? In our case, the problem immediately rears up: The available universe, for training and evaluation, is just the set of linked commits C_{fl} (except for our APACHE evaluation sample). What we would like to do is train on a biased sample, and evaluate on an unbiased sample. Given that all we have is a biased dataset C_{fl} to begin with, how are we to evaluate the effect of the bias? Our approach is based on sub-sampling. Since all we have is a biased set of linked samples, we test on all the linked samples, but we train on a linked sub-sample that has systematically enhanced bias with respect to the bug features. We consider both severity and experience. We do this in two different ways.

1. First we choose the training set from just one category, for example, train on only the critical linked bugs, and evaluate the resulting "super-biased" predictor on all linked bugs. We can then judge if the predictions provided by the super-biased predictor reflects the enhanced bias in the training set. We repeat this experiment with the training set drawn exclusively from each severity category. We also conduct this experiment choosing biased training sets based on experience.

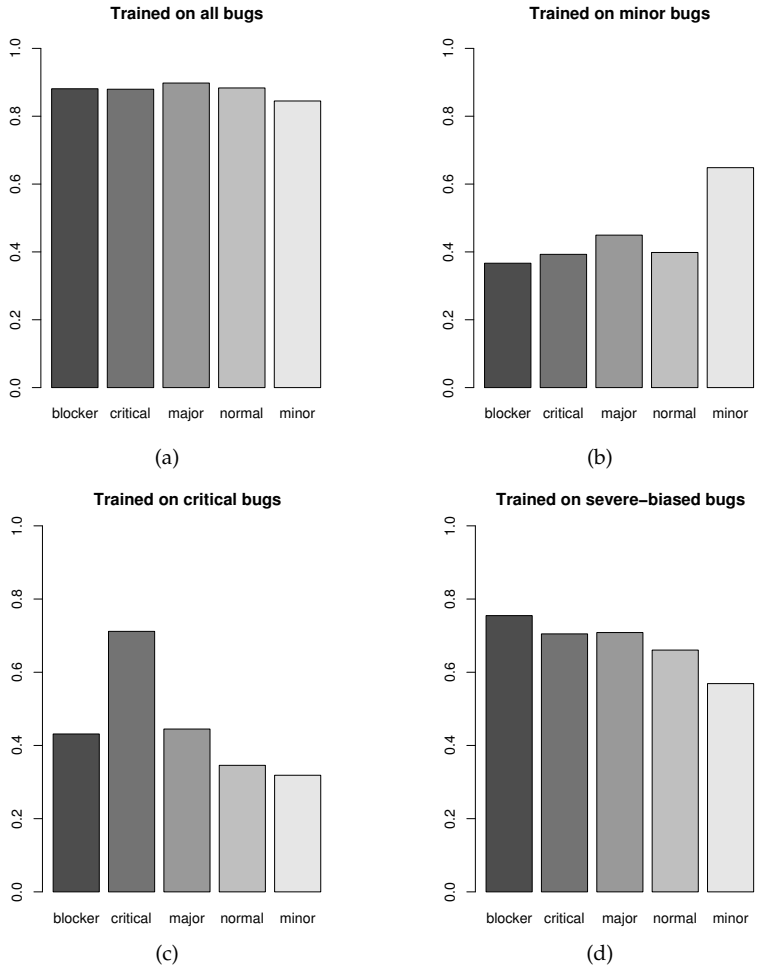


Figure 9.4: Recall of BUGCACHE for ECLIPSE when trained on all fixed bugs (a), only "minor" fixed bugs (b), only "critical" bugs (c), and a dataset biased towards more severe bugs (d)

2. Second, rather than training on bugs of only one severity level, we train on bugs of all severities, but chosen in a way to preserve but exaggerate the observed bias in the data. In our case, we chose a training sample in proportions that accentuates the slope of the graph in Figure 9.4 (rather than focusing exclusively on one severity category).

Finally, we train on all the available linked bugs, and evaluate the performance. For our evaluation, we implemented BUGCACHE as described in Kim *et al.* [Kim et al., 2007] along with methods used to gather data [Śliwerski et al., 2005] used by BUGCACHE. First, blame data is gathered for all the known bug fixing commits, i.e., $blame(C_{fl})$. These are considered bug introducing commits, and the goal of BUGCACHE is to predict these as soon as they occur, roughly on a real-time basis. BUGCACHE essentially works by continuously maintaining a cache of the most likely bug locations. We scan through the recorded history of commits, updating the cache according to a fixed set of rules described by Kim *et al.*, omitted here for brevity. A hit is recorded when a bug introducing commit is encountered in the history and is also discovered in the cache, and a miss if it is not in the cache. We implemented this approach faithfully as described by Kim *et al.* [Kim et al., 2007]. We use a technique similar to Kim *et al.* [Kim et al., 2006b] to identify the fix inducing changes, but use `git blame` to extract rework data, whereas `cvs annotate`. `git` tracks code copying and movements, which CVS and SVN do not, and thus provides more accurate blame data. For each of the types of bias, we evaluated the effect of the bias on BUGCACHE by systematically choosing a super-biased training set $B_t \subset B_{fl}$ and examining the effects on the predictions made by BUGCACHE. We run BUGCACHE and record hits and misses for all bugs in B_{fl}.

We use the recall measure, essentially the proportion of hits overall. We report findings when evaluating BUGCACHE on the ECLIPSE dataset. As a baseline, we start by training and evaluating BUGCACHE with respect to bug severity on the entire set B_{fl} for ECLIPSE. Figure 9.4a shows the proportion of bugs in B_{fl} that are hits in BUGCACHE. The recall is around 90% for all severity categories.

Next, we select a superbiased training set B_t and evaluate BUGCACHE on B_{fl}. We do this by recording hits and misses for all bugs in B_{fl}, but only updating the cache when bugs in B_t are missed. If B_t and B_{fl} share locality, then performance will be better. For each of the severity levels, we set B_t to only the bugs that were assigned that severity level and evaluated the effect on BUGCACHE. Figure 9.4b shows the recall of BUGCACHE when B_t included only the minor bugs in B_{fl} from ECLIPSE and Figure 9.4c shows the recall when b_t included only critical bugs. It can be seen that the recall performance also shows a corresponding bias, with better performance for the minor bugs. We found that BUGCACHE responds similarly when trained with superbiased training sets drawn exclusively from each severity class, except for the normal class: We suspect this may be because normal bugs are more frequently co-located with bugs of other severity levels, whereas critical bugs, for example, tend to co-occur with other critical bugs. We also evaluated BUGCACHE with less extreme levels of bias, for instance, when B_t was composed of 80% of the blocker bugs, 60% of the critical, etc. The recall for this scenario is depicted in Figure 9.4d. The bias towards higher severity in bug hits still existed, but was less pronounced than in Figure 9.4c.

From this trial, it is likely that the Bug Type: Severity bug feature bias in B_{fl} affects the performance of BUGCACHE. We also evaluated the effect of bias in a bug process feature, for example, experience, on BUGCACHE. We divided the bugs in B_{fl} into those fixed by experienced project members and those fixed by inexperienced project members by splitting around the median experience of the closers for all bugs in B_{fl}. When BUGCACHE was trained only on bugs closed by experienced closers, it did poorly at predicting bugs closed by inexperienced closers and vice versa in ECLIPSE. This suggests that experience-related feature bias can also affect the performance of BUGCACHE. In general, BUGCACHE predicts best when trained on a set with bug feature bias similar to the test set.

Does bug feature bias affect the performance of prediction systems? The above study examines this using a specific model, BUGCACHE, with respect to two types of bug feature bias. We made two

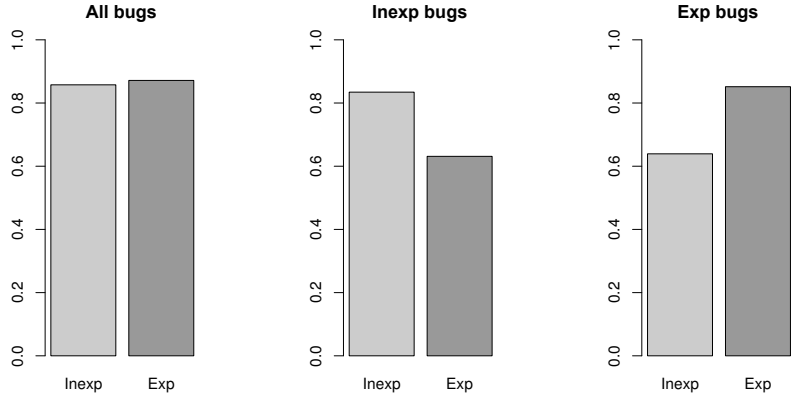


Figure 9.5: Recall of BUGCACHE for ECLIPSE when trained on all bugs (a), and only bugs marked fixed by inexperienced (b) or experienced (c) people

observations: (i) If you train a prediction model on a specific kind of bug, it performs well for that kind of bug, and less well for other kinds. (ii) If you train a model on a sample including all kinds of bugs, but which accentuates the observed bias even further, then the performance of the model reflects this accentuation. These observations cast doubt on the effectiveness of bug prediction models trained on biased models.

9.4 Analysis of Commit Feature Bias

The manual annotation effort for the APACHE project presented in the previous chapter indicates that many bug fixes are not identified in the commit logs, and thus are completely invisible to the automated linking tools used to extract bug fix data. Thus the linked bug fix commits are a sample of the entire group of commits. However, samples

thus extracted have been central to many research efforts. The natural question is: Is this sample representative, or biased?

We seek to test for the two kinds of bias *bug feature bias*, whereby only fixes to certain kinds of bugs are linked, and *commit feature bias*, whereby only certain types of commits are linked. With access to the entire set of fixed bugs, and the subset of linked bugs, we could check for bug feature bias (see previous sections). Unfortunately, analyzing commit feature bias is usually not possible since C_f is only partially known In the previous chapter, we fully annotated a temporal sample of APACHE commits. This verified and fully annotated APACHE evaluation sample dataset allows us to check for commit feature bias. Remember, commit features are properties of the file and its revision history, such as size, complexity, authorship, etc. These are critical properties that have been studied in dozens of papers that test theories of bug introductions; they are also the features used for bug prediction. So it is important to test for commit feature bias and evaluate its impact. In this section, we describe some findings related to commit feature bias and in the next section we describe the effect of commit feature bias on BUGCACHE.

We remind the reader that our APACHE evaluation sample size (despite the time and effort required to gather even that much) is not big enough to realistically expect to find statistically significant support for answers to the questions discussed in this and the next section. Therefore, we do not introduce new hypotheses but analyze a set of questions related to Hypotheses 3.2 and 3.3. However, there are some takeaways: We do find statistical support for the answer to one question, and we do find some anecdotal answers for the other questions. Furthermore, actual bias along any of the lines discussed here would have a highly deleterious effect on the external validity of theories tested using only the linked data. Most importantly, we hope to convince the reader that such studies are important and need to be repeated and conducted at larger scales.

The first question arises naturally from the fact that there are different individual developers, who may have different attitudes towards linking.

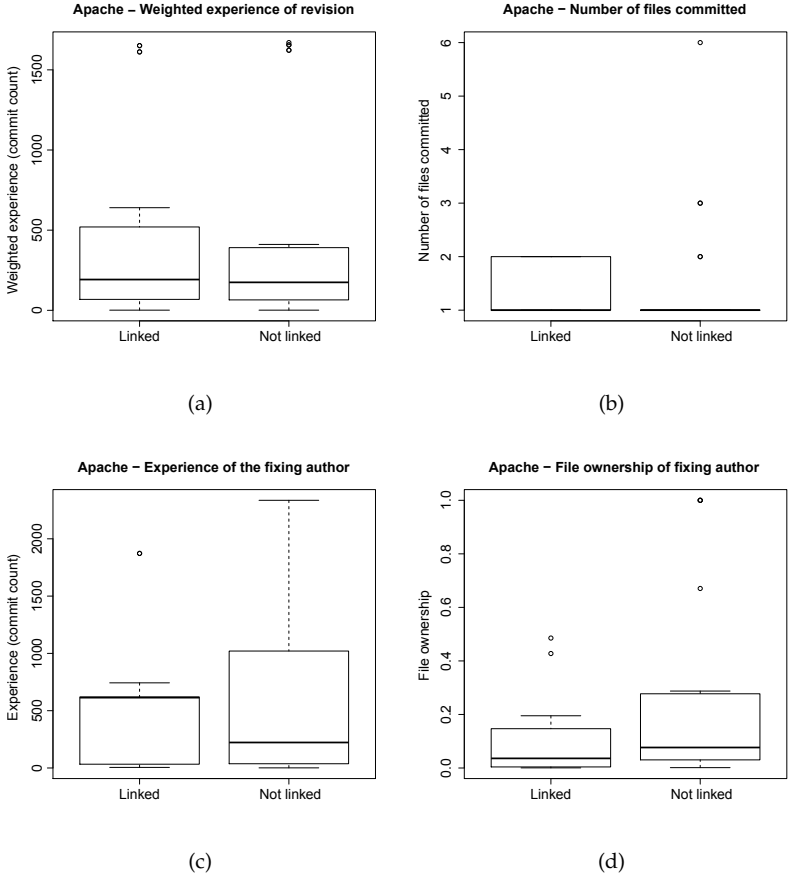


Figure 9.6: Commit feature bias: Weighted experience of the original authors of the fix-inducing code (a); number of files changed in the bug fix (b); experience of the author committing the bug fix (c); proportion of fixed files owned by bug fix author at the time of the bug fix (d)

Do different developers show significantly different linking behavior? Table 9.2 lists the APACHE developers and their linking behavior (anonymized). For each developer (a, b, c ...) we show the number they linked and the number not linked. A casual look shows differences, for example, that developer *b* is a good linker, whereas developer *a* does not link any. Even with the low numbers in the table, we can conclude via Fisher's exact test that the developers' linking proportions are significantly different ($p \simeq 0.002$).

Table 9.2: APACHE developers and their linking behaviors (anonymized, "#" = number of)

Name	#Linked	#Not linked	Name	#Linked	#Not linked
a	0	6	b	10	5
c	1	1	d	11	8
e	0	3	f	0	1
g	0	3	h	0	5
i	2	7	j	0	3
k	0	2	l	0	1
m	0	2	n	0	1
o	0	1	p	1	0
q	4	0	Total	26	52

We now hypothesize several different specific possible motivational theories of linking behavior. In several cases, there was a visually apparent signal, in boxplots, albeit none that were statistically significant. The results are shown in Figure 9.6. We list them below, but we caution the reader to interpret all these findings as at best anecdotal. However, it is important to bear in mind that actual bias influenced by any of the processes hypothesized below would be very damaging to the external validity of theories tested solely on the linked data.

Does the experience of the author(s) whose code is being fixed influence linking behavior? We hypothesized that the quest for greater reputation might incentivize people to link fixes when the code under

repair belonged to an experienced (and thus more reputable) person. We measured the fixed code's "author reputation" as the geometric of the prior commit experience of everyone who contributed to the fixed code. The boxplot (Figure 9.6a) is weakly suggestive that fixes made to code with more experienced authorship are more likely to be linked.

Does the number of files involved in the bug fix matter? If more files are repaired in a bug fix, perhaps the fix is more "impactful"; this might motivate the developer to more carefully document the change. In fact, the boxplot (Figure 9.6b) is suggestive that this might be the case, with all the unlinked fixes being single-file fixes.

Are more experienced bug fixers more likely to link? We might expect that more experienced developers behave more responsibly. We measure experience as the number of prior commits. The boxplot (Figure 9.6c) suggests support for this theory, with a noticeably higher median for the linked case.

Are developers who "own" a file more likely to link bug fixes in that file? One might expect that people fixing bugs in their own files are more likely to behave responsibly and link; on the other hand, there is an anti-social reputation-preserving instinct that suggests that they may be less likely to link. We measure ownership as the proportion of lines in the file authored by the bug fixer. Indeed, the boxplot (Figure 9.6d) visually supports the "anti-social" theory.

9.5 Effects of Commit Feature Bias

The analysis in the previous section shows that the extent of bias in the data is significant and that the effort of finding the ground truth (e.g., through manual annotation of a dataset as discussed in Chapter 8) leads to important insights. But do those insights translate into practical impact? In this section we investigate the impact of approaching the ground truth in terms of changes in the accuracy of the award-

winning BUGCACHE algorithm [Kim et al., 2007]. To that end, we repeat our experiment showing the impact of bias using APACHE data in the previous sections. Specifically, we departed from two different datasets: The first dataset (called A below) contained all 1 576 bugs introduced in the APACHE 2.0 branch. The second one contained the additional 65 bugs found by Justin as discussed in Chapter 8 (called J). Table 9.3 shows the resulting accuracies for training and predicting on each combination of these two datasets.

Table 9.3: BUGCACHE prediction quality

Learning set	Test set	Accuracy	95% Confidence interval
A	A	0.875	[0.858, 0.890]
A	A ∪ J	0.870	[0.852, 0.885]
A	J	0.738	[0.620, 0.830]
A ∪ J	A	0.878	[0.860, 0.893]
A ∪ J	A ∪ J	0.874	[0.857, 0.889]
A ∪ J	J	0.785	[0.670, 0.867]

Consider training on the extracted data A and predicting on the same data. This provides a baseline accuracy of 0.875. If the prediction is, however, performed on the dataset representing ground truth for the period of manual annotation A ∪ J then the accuracy falls to 0.870. We accede that due to the limited manually annotated period the difference—like all the differences in the table—is not significant. But as the following shows that we can recognize a tendency. Alternatively, consider adding the manually annotated bugs to the training set (i.e., training on A ∪ J). In each possible prediction target (i.e., A, J, and A ∪ J) we find that the availability of the additional information actually leads to an improvement in prediction accuracy. This is especially impressive where the prediction target is A as it shows that the manually annotated bugs actually contain information relevant to the automatically extracted ones, helping BUGCACHE to find four additional bugs.

9.6 Threats to Validity

Generalizability of Results. The biases we observed may be specific to the processes adopted in the projects we considered; however, we did choose projects with varying governance structures, so the results seem robust.

Possible Linking Bias. As noted earlier, our study of bias effects may be threatened by highly specific (but rather unlikely) coincidences in bug occurrence and linking (as discussed in the previous chapters).

Performance of BUGCACHE. Sadly, we do not have an unbiased oracle to truly evaluate BUGCACHE's performance. Thus, BUGCACHE might be overcoming bias, but we are unable to detect it, since all we have is a biased linked sample to use as our oracle. First, we note that BUGCACHE is predicated on the locality and recency effects of bug occurrence. Second, the data indicates that there is a strong locality of bugs, when broken down by the severity and experience bug features. For BUGCACHE to overcome bug feature bias, bugs with features that are over-represented in the linked sample would have to co-occur (in the same locality) with unlinked bugs with features that are under-represented.

9.7 Concluding Discussion

In this section we introduced two types of bias in software engineering datasets: *bug feature bias*, where only the fixes of certain types of defects are reported, and *commit feature bias*, where only the certain kinds of fixes, or fixes to certain kinds of files, are reported. Either type of bias is highly undesirable. Unfortunately, we found evidence of bug feature bias in all datasets as hypothesized (Hypothesis 3.2), although we have lower levels of bug feature bias in the CSS projects investigated. Our experiments also suggest that bug feature bias affects the performance of the the award-winning BUGCACHE defect prediction algorithm (Hypothesis 3.3).

We then showed the sources and the extent of the commit bias of

the automatically extracted APACHE data in comparison to the manually annotated APACHE evaluation sample. Especially notable is the significant variation in linking behavior among developers, and the anecdotal evidence suggesting that bug fixing experience and code ownership play a role in linking behavior. We also showed that commit feature bias affects the performance of a bug prediction algorithm—BUGCACHE has strong tendencies to miss predictions if it is not trained on the ground truth (Hypothesis 3.3).

Summarizing, our work suggests that both types of bias exist in currently used datasets and that bias is a serious problem for empirical software engineering studies that rely on such data. Empirical software engineering researchers, therefore, should care about data quality. Looking forward, we ask what can be done about bias.

One possibility is the advent of systems like JAZZ that force developers to link commits to bugs and/or feature requests. However, experience in other domains suggests that enforcement provides little assurance of sufficient data quality [Zuboff, 1988]. In the next part, we therefore discover why not only empirical software engineering researchers but also software engineering practitioners should care about data quality in software engineering. Our findings may motivate practitioners to enforce better data quality and, therefore, reduce bias in datasets.

Part V

Why Should Practitioners Care about Data Quality in Software Engineering?

10

When Process Data Quality Affects Process Quality[1]

Analyzing data quality in software engineering, we found that all datasets investigated are affected by quality issues and we are able to link only a sample of fixed bugs to specific commits. Even worse, this sample of linked and fixed bugs is not representative of the set of all bugs since we found both bug feature bias and commit feature bias in all datasets. The BUGCACHE experiments showed that empirical software engineering results may be threatened by data quality issues. This raises two questions: "How can we counter these data quality issues?" and "Are software engineering practitioners affected by these issues?".

If we are able to evaluate effects of poor data quality on software engineering process quality or, even worse, on software product quality, we can answer both questions. Specifically, one may assume that, for instance, empty commit messages have a negative impact on bug fixing performance and, maybe, also on the number of future bugs.

[1]Parts of this chapter have already been published [Bachmann and Bernstein, 2010]

Such findings would have the potential to prompt practitioners to increase the quality of their software processes and the associated data quality. Therefore, in this and the next chapter we analyze the following hypotheses:

HYPOTHESIS 4.1: *Poor software engineering data quality influences the bug fixing process and bug fixing activities (i.e., performance of bug fixing).*

HYPOTHESIS 4.2: *Software engineering process quality (evaluated by process data quality) influences product quality (measured by number of bugs).*

First, we discuss the theoretical background of our evaluation. We then present several sub-hypotheses related to Hypothesis 4.1 and discuss our results and findings.

10.1 Evaluation Procedure and Theory

10.1.1 Measurement of Process Quality

In recent publications several approaches to measure and ensure quality in processes are discussed (e.g., [Ashrafi, 2003; Harter and Slaughter, 2000; Kan, 2002]). For our evaluation, we use a statistical approach that measures the quality of data provided by software engineering tools and systems used in these processes (see Chapter 3). In particular, we make use of software engineering process data quality and characteristics measures (as presented in Chapter 6). These measures support an evaluation of data quality and characteristics of software engineering data considering all tools used in the process and their data. For this, we compare the values of the measures including their changes over time to uncover possible relations. Specifically, we compute the measures on weekly data frames for each of the projects and

use correlation values to unearth possible relations between the measures.

10.1.2 Calculation of Correlation Values

To analyze possible relations between the measures, we compute correlation values between them. For the correlation we use the Kendall tau (τ) rank correlation coefficient [Kendall, 1938]. In contrast to other correlation coefficient metrics (e.g., Spearman or Pearson), this correlation coefficient has *substantial advantages*. Above all, the Kendall tau rank correlation coefficient

1. makes no assumptions about the particular nature of the relationship between the variables (linear relationship not required),
2. does not require a normal distribution of the data,
3. does not require equidistance of the values, and
4. has a high robustness against outliers.

In addition, it allows an easy interpretation of values which lie between -1 and 1. Please note that *the correlation values of Kendall tau cannot be meaningfully compared to other rank correlation coefficients values* such as Spearman's ρ as it usually generates lower correlation values. Since the interpretation of the rank correlation values is not standardized, we used the following interpretation schema [Cohen, 1988]:

$0.1 \leq |\tau| < 0.3$ **weak correlation**
$0.3 \leq |\tau| < 0.5$ **moderate correlation**
$0.5 \leq |\tau| \leq 1.0$ **strong correlation**

To test the τ correlation values for significance, we calculate and report the two-sided p-values (t-test).

10.2 Interplay of Data Quality and Data Characteristics

In Chapter 7, we showed that the quality and characteristics of process data varies across projects. Therefore, we calculated the measures on weekly data frames and and found that data quality and characteristics measures vary over time. To illustrate these changes, Figure 10.1 shows the computed quality measures for ECLIPSE on weekly frames over a period of several years. Analyzing the figure, an interesting question arises: Why do these values change over time?

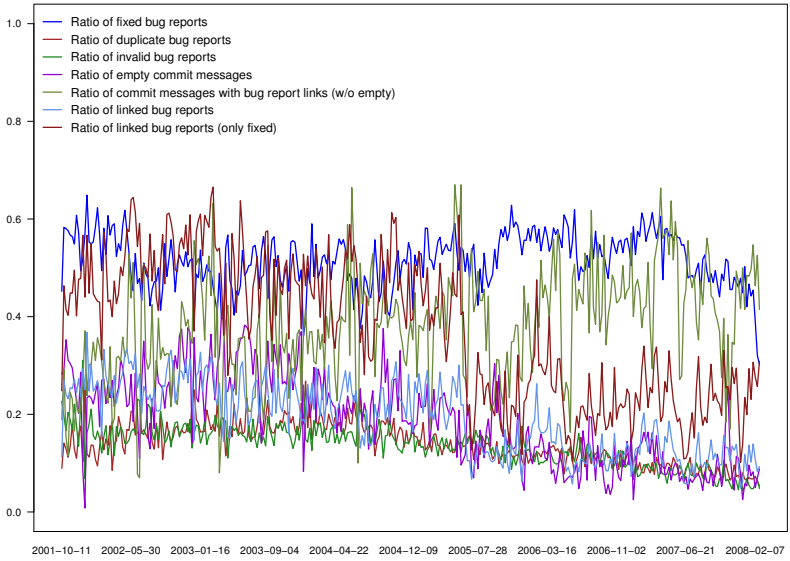


Figure 10.1: Data quality measure values for ECLIPSE (weekly frames)

Possible reasons may be project-external influences (changes of user habits, growing user community, etc.) as well as changes in the project (e.g., new developers or release dates). In Chapter 9 we analyzed whether we have bug feature bias by release date features and found

no evidence to support this hypothesis. Therefore it seems feasible, due to similar value changes over time in many of the characteristics and quality measures, that the phenomena they measure are connected. For example, empty commit messages are undesirable due to a lack of information about a commit's reason and therefore may influence the performance of bug fixing evaluated by data quality measures.

In this section, therefore, we analyze and discuss several subhypotheses to uncover possible relations between the measures calculated on weekly data frames.

10.2.1 Impact of Empty Commit Messages

As already discussed, we hypothesize that empty commit messages are undesirable and therefore may have an impact on bug fixing processes.

HYPOTHESIS 4.1.1: *Empty commit messages have an impact on bug fixing process quality.*

The problem of empty commit messages arises mainly in the ECLIPSE and BSZKB#2 datasets where lots of commit messages are empty. In all other datasets, this ratio is far below 1% (see Table 7.1 in Chapter 7). Therefore, we test this hypothesis on the ECLIPSE and BSZKB#2 project data only. Table 10.1 lists the τ correlation coefficient values between the "Ratio of empty commit messages" measure and the other quality measures. The two-sided p-values are also listed in the table.

For ECLIPSE, the τ values show a moderate to strong positive correlation between the "Ratio of empty messages" and the proportion of duplicate, invalid, and linked bug reports, while the "Ratio of fixed bug reports" exhibits only a weak correlation. For BSZKB#2 we found no significant correlation values and all τ values have high p values. Considering the ECLIPSE results only, we can conclude that the more empty messages there are, the more duplicate and invalid bug reports

Table 10.1: Kendall τ correlations for "Ratio of empty messages" in ECLIPSE and BSZKB#2

	ECLIPSE		BSZKB#2	
Quality measure	**τ value**	**p value**	**τ value**	**p value**
Ratio of fixed bug reports	-0.096	$< 8.652 \cdot 10^{-3}$	-0.111	$< 9.691 \cdot 10^{-2}$
Ratio of duplicate bug reports	0.481	$< 2.22 \cdot 10^{-16}$	-0.015	$< 8.4771 \cdot 10^{-1}$
Ratio of invalid bug reports	0.481	$< 2.22 \cdot 10^{-16}$	N/A	N/A
Ratio of commit messages with bug report links (w/o empty)	-0.228	$< 5.397 \cdot 10^{-10}$	0.051	$< 4.0975 \cdot 10^{-1}$
Ratio of linked bug reports	0.475	$< 2.22 \cdot 10^{-16}$	0.161	$< 1.541 \cdot 10^{-2}$
Ratio of linked bug reports (only fixed bug reports)	0.488	$< 2.22 \cdot 10^{-16}$	-0.005	$< 9.385 \cdot 10^{-1}$

we have. On the other hand, there is also a beneficial effect, due the positive correlation to linked bug reports. Unfortunately, we were able to test this hypothesis with two datasets only—all other datasets have very low ratios of empty messages. Interestingly, only ECLIPSE supports the hypothesis. One may assume that empty commit messages are biased in the BSZKB#2 projects such that only the less relevant program changes are committed without message. Unfortunately, due to lack of information (remember, we need the ground truth to test for commit feature bias), we are not able to test the BSZKB#2 dataset for this kind of commit feature bias and therefore are not able to support our assumption.

As already mentioned in Sub-Section 10.1.2, the τ correlation values are usually lower than other correlation values. For illustration

purposes, Table 10.2 contains the Kendall τ and the Spearman's ρ correlation values.

Table 10.2: Comparison of Kendall τ and Spearman ρ correlation values for "Ratio of empty messages" in ECLIPSE

Quality measure	τ value	ρ value
Ratio of fixed bug reports	−0.096	−0.142
Ratio of duplicate bug reports	0.481	0.672
Ratio of invalid bug reports	0.481	0.671
Ratio of commit messages with bug report links (w/o empty)	−0.228	−0.337
Ratio of linked bug reports	0.475	0.671
Ratio of linked bug reports (only fixed bug reports)	0.488	0.691

10.2.2 Developer Workload and Linking Ratio

Usually, bugs are reported by professional testing engineers (CSS) or users (OSS) who discover an unwanted behavior of the system during testing activities or its use. Particularly after new releases, more bugs are found and reported. This also leads to a varying number of bug reports over time to be taken care of by developers. Given that developer work time is a limited resource, we hypothesize that when the workload—measured by number of bug reports to fix—goes up, developers will only be able to attend to a smaller fraction of them—measured by bug reports that get fixed and linked to commits.

HYPOTHESIS 4.1.2: *The higher the developer workload, the fewer bug reports get fixed and linked in commits.*

MOZILLA and GNOME have moderate negative correlations between developer workload and bug fixing ratio. This means, that in

these projects the more bugs a developer has to fix (on average), the fewer bugs actually get fixed. Conversely, both projects have a high bug duplicate ratio of about 33% (see Table 7.1), which may have an effect such as duplicates usually never getting fixed. OPENOFFICE also has a weak negative correlation with a similar finding, namely that this project has a bug duplicate ratio above average. ECLIPSE and BSZKB#2 have weak positive correlations. Both projects have positive correlations (ECLIPSE $\tau = 0.574$, BSZKB#2 $\tau = 0.181$) between "Average commits per bug report" and "Ratio of linked bug reports". This means that the more bugs get reported (and therefore the developer's workload and the proportion of bugs to commits increase), the higher the fixing ratio. In contrast, MOZILLA and GNOME also have high correlations between "Average commits per bug report" and "Ratio of linked bug reports" but have negative correlations between developer workload and "Ratio of fixed bug reports". All other projects have no significant correlations in the data (left part of Table 10.3).

Table 10.3: Kendall τ correlations between "Average bug reports per developer" and "Ratio of fixed bug reports" or "Ratio of linked bug reports"

	"Ratio of fixed bug reports"		"Ratio of linked bug reports"	
Project	**τ value**	**p value**	**τ value**	**p value**
APACHE	0.032	$< 5.103 \cdot 10^{-1}$	0.021	$< 6.842 \cdot 10^{-1}$
ECLIPSE	0.160	$< 1.398 \cdot 10^{-5}$	−0.407	$< 2.22 \cdot 10^{-16}$
GNOME	−0.449	$< 2.22 \cdot 10^{-16}$	−0.209	$< 7.215 \cdot 10^{-11}$
NETBEANS	−0.080	$< 1.531 \cdot 10^{-2}$	0.005	$< 8.751 \cdot 10^{-1}$
OPENOFFICE	−0.261	$< 1.382 \cdot 10^{-14}$	0.132	$< 1.053 \cdot 10^{-4}$
MOZILLA	−0.412	$< 2.22 \cdot 10^{-16}$	−0.550	$< 2.22 \cdot 10^{-16}$
BSZKB#1	0.022	$< 6.852 \cdot 10^{-1}$	0.109	$< 5.477 \cdot 10^{-2}$
BSZKB#2	0.327	$< 1.783 \cdot 10^{-6}$	0.333	$< 8.007 \cdot 10^{-7}$

Regarding the correlations between developer workload and "Ratio of linked bug reports", ECLIPSE, GNOME, and MOZILLA support the hypothesis that the higher the developer's workload, the lower the

"Ratio of linked bug reports". In contrast, OPENOFFICE and interestingly both ZKB projects have a weak to moderate positive correlation (right part of Table 10.3). These three projects have fewer developers compared to the other projects, which may have an impact on this correlation.

10.2.3 Bug Reporter Experience and Fixed Bugs

In OSS projects it is a usual practice to release alpha and beta versions of software and perform testing by users. In the CSS projects investigated, in contrast, a professionalized testing is performed by a few testers (see discussion in Section 3.1.2). These testers follow a defined test procedure composed of test cases, test scenarios, and test data. Whereas in professionalized testing only a few people report bugs, in OSS alpha and beta testing many people use/test a new release and report bugs. Therefore, in many OSS projects, most users only report a few bugs in their life, potentially leading to a lower quality of the bug reports due to lacking experience. A bug report of poor quality may lower the probability of it being fixed, as the developer may not understand it.

HYPOTHESIS 4.1.3: *The more experienced the bug reporters, the higher the chance of bug reports being fixed.*

MOZILLA, GNOME, ECLIPSE, and BSZKB#2 have a moderate to strong positive correlation; APACHE has a weak positive correlation. All these projects support the hypothesis with very low p-values that the more bugs a reporter reports, the higher the linking ratio. Only NETBEANS and OPENOFFICE weakly reject the hypothesis (Table 10.4). Despite these two counterexamples, we believe that the evidence of the other projects suggests that the more bugs somebody reports, the better the quality of these reports, and, therefore, the higher the likelihood of a developer understanding and fixing the problem. This finding justifies methods for enhancement of bug report quality (e.g., Bettenburg *et al.* [Bettenburg et al., 2007a,b, 2008]) and the use of better tool support.

Table 10.4: Kendall τ correlations between "Average bug reports per bug reporter" and "Ratio of fixed bug reports"

Project	τ value	p value
APACHE	0.123	$< 1.855 \cdot 10^{-2}$
ECLIPSE	0.342	$< 2.22 \cdot 10^{-16}$
GNOME	0.427	$< 2.22 \cdot 10^{-16}$
NETBEANS	-0.118	$< 3.611 \cdot 10^{-4}$
OPENOFFICE	-0.192	$< 1.577 \cdot 10^{-8}$
MOZILLA	0.604	$< 2.22 \cdot 10^{-16}$
BSZKB#1	0.054	$< 3.305 \cdot 10^{-1}$
BSZKB#2	0.417	$< 6.492 \cdot 10^{-9}$

10.2.4 Bug Report Discussion and Linking Ratio

Important bugs usually receive a high level of attention by the community and, therefore, many users and developers make use of the discussion function of the BTS. We hypothesize that the more attention a bug has (measured by number of comments), the higher the likelihood of a developer mentioning the report in the commit message of the bug fix.

HYPOTHESIS 4.1.4: *The more people discuss a bug report, the higher the likelihood of that bug report becoming linked.*

This hypothesis is supported by the strong correlation in GNOME and the weak to moderate correlations of all other projects except one (Table 10.5). ECLIPSE does not support or contradicts this hypothesis with $\tau = -0.026$. We believe that a high level of attention by the community acts as an incentive to developers for a better documentation of their work (which includes the linking of bug fixes to bug reports). In contrast, without attention from the community, a developer has fewer benefits in linking bug fixes and, therefore, might not be so strict.

Table 10.5: Kendall τ Correlations between "Average Comments per bug report" and "Ratio of linked bug reports"

Project	τ value	p value
APACHE	0.255	$< 3.682 \cdot 10^{-7}$
ECLIPSE	−0.026	$< 4.702 \cdot 10^{-1}$
GNOME	0.605	$< 2.22 \cdot 10^{-16}$
NETBEANS	0.323	$< 2.22 \cdot 10^{-16}$
OPENOFFICE	0.112	$< 1.025 \cdot 10^{-3}$
MOZILLA	0.297	$< 2.22 \cdot 10^{-16}$
BSZKB#1	0.227	$< 5.909 \cdot 10^{-5}$
BSZKB#2	0.348	$< 4.148 \cdot 10^{-7}$

10.3 Threats to Validity

Generalizability of Results. Software engineering tools and processes vary in different projects and, therefore, our findings based on the projects investigated may not generalize to other projects. However, we analyzed often-used and well-known OSS projects. Therefore, it is reasonable to conclude that the selected projects are no exception in OSS engineering. On the other hand, we were only able to analyze two CSS projects, both provided by the Zurich Cantonal Bank. Therefore, we acknowledge threats in generalizing these results to other CSS projects.

Correlation and Causality. It is important to note that all our explorations are based on correlations. Hence, we cannot make definitive statements on causality.

10.4 Concluding Discussion

In the previous chapters, we analyzed the impact of poor data quality in software engineering on empirical software engineering research.

We found that considering data quality concerns is crucial for research in empirical software engineering. In this chapter, we elaborated part of Research Question 4 and showed why practitioners should care about the quality of their processes and data. Specifically, we analyzed the hypothesis that data quality issues in software engineering influence the performance of bug fixing activities. To test this hypothesis, we used the software engineering datasets investigated and calculated our data quality and characteristics measures for each week and project. We then calculated Kendall tau rank correlations based on these measures. In particular, we showed that data quality and characteristics issues affect each other. For instance, the proportion of empty commit messages in ECLIPSE correlate with bug report quality. In addition, we showed that the more active a community is in its communication—such as the discussion and commenting bug reports—the better the linking ratio provided by the developers is. These findings have the potential to prompt practitioners to increase the quality of their software processes and its associated data quality.

In the next chapter, we extend our evaluation and analyze whether software engineering data quality affects software data quality.

11

When Process Data Quality Affects the Number of Bugs[1]

In the previous chapter, we showed that data quality issues have an effect on bug fixing activities and data quality. These findings have the potential to prompt practitioners to increase the quality of their software processes and their associated data quality. In this chapter we extend our evaluation and analyze if process data quality influences product quality:

HYPOTHESIS 4.2: *Software engineering process quality (evaluated by process data quality) influences product quality (measured by number of bugs).*

Again, we define several sub-hypotheses and make use of our data quality and characteristics measures to evaluate these hypotheses. In addition, we define software product quality as "number of bugs", allowing us to calculate correlation values between the measures and product quality.

[1]Parts of this chapter have already been published [Bachmann and Bernstein, 2010]

11.1 Evaluation Procedure and Theory

To evaluate the hypotheses in this chapter, again we make use of Kendall tau rank correlations (see discussion in Section 10.1.2). This section, therefore, only contains complementary information not discussed in the previous chapter.

11.1.1 Measurement of Software Product Quality

The evaluation and measurement of software product quality is a widely explored field with many approaches. Already in 1978, for instance, Cavano and McCall presented a framework to measure software quality [Cavano and McCall, 1978]. They defined the following software quality dimensions and factors:

- Product revision: maintainability, flexibility, and testability
- Product transition: portability, reusability, and interoperability
- Product operations: correctness, reliability, efficiency, integrity, and usability

We acknowledge that software engineering changed over the past few years and, therefore, many approaches for software quality management and measurement were newly developed. Nonetheless, we believe that this framework still defines the most important factors for software quality evaluation, although the labels have changed somewhat.

In this chapter, we mainly focus on the product operations aspect, i.e., quality factors affecting the users. These are typically reported as bugs whenever they exhibit unwanted behavior (e.g., code is wrong, too slow, crashes, etc.). Luckily, it is very easy to evaluate the number of bugs if we have access to a BTS.

Measuring product quality by number of bugs, we asked ourselves what kind of bugs we should consider: all bugs or only post-release bugs. Based on Research Question 4 and Hypothesis 4.2, we should consider all bugs which were released and available to the public in

any release of the software. Irrelevant of the version in which version a bug was found, it was implemented and released without being noticed by developers. Consequently, we define the product quality in this thesis as the number of bugs reported over time (pre- and post-release), which is common practice (see [Kan, 2002]).

11.1.2 Time-Shifted Correlations

Some effects may only appear after a time delay. Bugs are usually discovered some time after the introducing commit. Hence, we compute not only the correlation between the measures and the number of bugs within the same week but also calculate time-shifted correlations to uncover time-shifting effects. Essentially, we calculate the correlation between the measures at time $t = 0$ and the number of bugs reported at time $t \pm 0 \ldots 50$ weeks. For all time constrained values we use the week of reporting or committing as the relevant time information.

11.2 Influence of Process Quality on Product Quality

With Hypothesis 4.2, we try to evaluate the impact of process (data) quality on product quality. For this, we present several sub-hypotheses, provide a contextual discussion, and discuss the results and findings based on the datasets investigated. Unfortunately, we found no significant product quality correlations in APACHE and BSZKB#2 for any of the metrics. Therefore, we omit a further discussion of the results for these two projects.

11.2.1 Poor Process Data Quality Today – More Bugs Tomorrow?

We believe that an impact of process quality on product quality may be time-shifted, for example, when the process quality drops then more bugs are introduced later due to poor documentation quality (e.g., missing linking information).

HYPOTHESIS 4.2.1: *Today's poor data quality has an effect on the number of bugs reported in the future.*

Hence, we test the hypothesis using time-shifting correlations. Figure 11.1 illustrates the time-shifted τ correlation values between our process data quality measures and the number of bugs. We calculated for each project the correlation between quality measures at $t = 0$ and number of bugs at $t \pm 0 \ldots 50$ weeks. Interestingly, most correlations have their maximum values at $\Delta t = 0$ (except for OPENOFFICE). We backed up this finding by statistics and confirmed that we do not seem to have any time-shifting effects in these datasets and have to reject Hypothesis 4.2.1. We find this surprising given the findings presented in the previous chapters.

11.2.2 The Effect of Duplicate and Invalid Bugs on Product Quality

Particularly in OSS projects, all users of a system are allowed to report bugs (see discussion in Section 3.2.3). Hence, many duplicate and invalid bug reports are stored in the BTS (as shown in Table 7.1) and need the attention of developers. Whether duplicate bug reports are valuable or not is a controversial discussion in current research. For instance, Bettenburg *et al.* believe that duplicates provide additional information to a specific problem and are not harmful *per se* [Bettenburg et al., 2008]. Nevertheless, duplicate and invalid bug reports increase the needed effort by developers to fix a specific problem and raise the risk that multiple developers work on similar problems. This

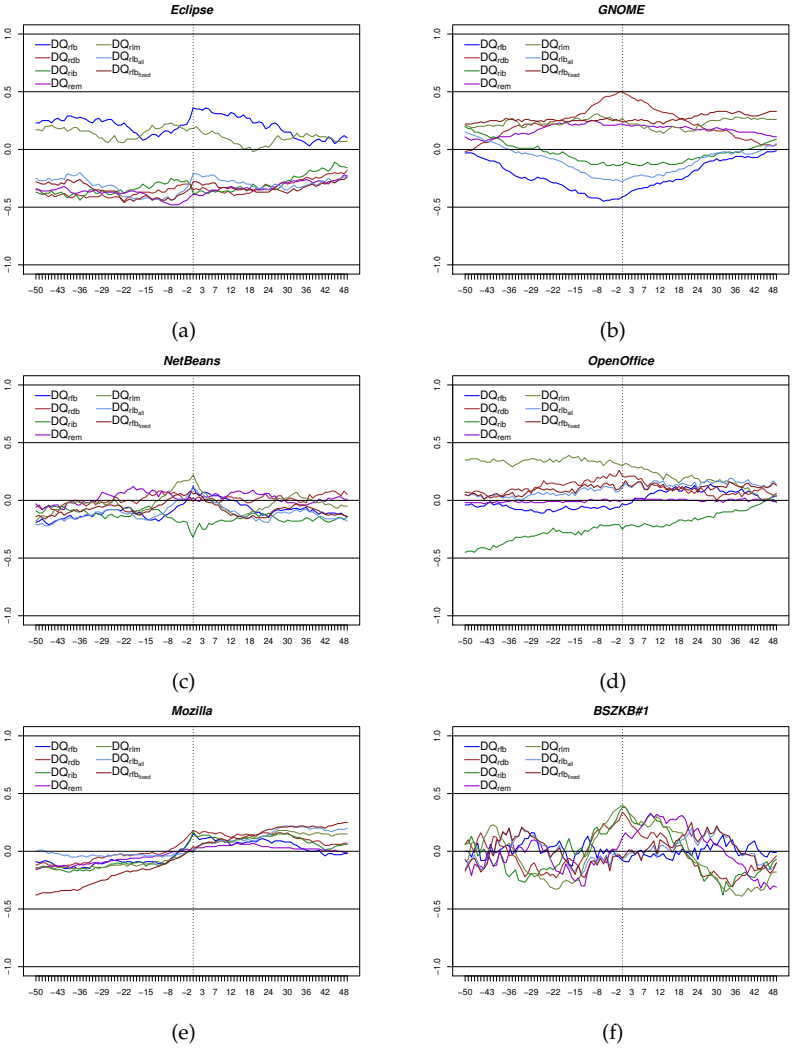


Figure 11.1: Time-shifted correlations between process quality measures and number of bugs

could give rise to two effects: (1) Developers have less time to carefully implement bug fixes and feature requests, and (2) synchronous work of multiple developers on the same problem may lead to conflicts. Therefore, we believe that duplicate and invalid bug reports may influence the number of (future) bugs.

HYPOTHESIS 4.2.2: *The proportion of duplicate and invalid bugs correlates with the number of bugs.*

Analyzing Figure 11.1, we see that the maximum values lie at $\Delta t = 0$, which we also verified statistically. Therefore, we present the τ and p values for $\Delta t = 0$ only (Table 11.1).

Table 11.1: Kendall τ correlations between "Number of bug reports" and "Ratio of duplicate bug reports", $\Delta t = 0$

Project	τ **value**	p **value**
ECLIPSE	-0.304	$< 2.22 \cdot 10^{-16}$
GNOME	0.640	$< 2.22 \cdot 10^{-16}$
NETBEANS	0.086	$< 8.891 \cdot 10^{-3}$
OPENOFFICE	0.310	$< 2.22 \cdot 10^{-16}$
MOZILLA	0.169	$< 1.696 \cdot 10^{-9}$
BSZKB#1	0.424	$< 1.147 \cdot 10^{-11}$

GNOME, BSZKB#1, and OPENOFFICE have a moderate to strong and MOZILLA a weak positive correlation and, therefore, support Hypothesis 4.2.2. Only ECLIPSE contradicts the hypothesis with a moderate negative correlation ($\tau = -0.304$). We believe that in the ECLIPSE project duplicate bug reports are not a substantial problem and provide additional information to the developers, as proposed by Bettenburg *et al.* Otherwise, a negative correlation would not be meaningful. For the correlations with invalid bug reports, we got similar correlations. In summary, invalid and duplicate bug reports seem to engage developers more than they support the introduction of new bugs.

11.2.3 The Influence of Missing Links in Commit Messages

A VCS usually allows concurrent development and tracks all changes: all program code changes are associated with a developer and changetime. In addition, developers usually describe in the commit message why a change was done (see discussion in Chapter 3.2.2). In the bug fixing process, such commit messages ideally contain links to bug reports they fix. Hence, these changes are justified and later other developers can follow the rationale behind the changes. Clear reasoning is conductive to an efficient maintenance of the software by developers, because they are able to find specific commits much more quickly than without a meaningful commit justification (e.g., empty messages).

Consequently, we hypothesize that the better the program code changes (commits) are documented, the fewer bugs are implemented in future releases. For example, we can assume the higher the linking ratio of bug reports, the more bug fixes and commits are documented/justified and therefore the fewer bugs get introduced.

HYPOTHESIS 4.2.3: *The proportion of linked bugs correlates with the number of bugs.*

Again, we calculated the τ correlation values between the number of bugs and "Ratio of linked bug reports" for $t \pm 0 \ldots 50$ weeks. Unfortunately, we did not find any significant correlations for any Δt in any project and have to reject Hypothesis 4.2.3.

11.2.4 The Influence of Empty Commit Messages

For similar reasons discussed above, we believe that empty commit messages have an impact on the number of bugs. Empty commit messages provide no information about the commit's reason. Again, we hypothesize that a good documentation of program code changes allows a faster and less buggy maintenance and enhancement of a software system. We therefore believe that developers tend to implement

more bugs due to the missing information in commits.

HYPOTHESIS 4.2.4: *The proportion of empty commit messages correlates with the number of bugs.*

As already mentioned earlier, ECLIPSE and BSZKB#2 are the only projects with a high "Ratio of empty commit messages". Therefore, we calculated the correlations for these two projects only. Interestingly, we found an almost strong negative correlation of $\tau = -0.483$ with a two-sided p-value of $< 2.22 \cdot 10^{-16}$ at $t = -6$ for ECLIPSE. The negative correlation indicates that when the "Ratio of empty commit messages" sinks, the number of bug reports rises. Or in other words: If we have fewer empty commit messages, we get more bug reports. The time-shift of $t = -6$ signifies that a high number of bug reports correlates with a high number of empty messages six weeks later. This is exactly the opposite of what we hypothesized. Hence, it could be hypothesized that an increase in bug reports leads to "overworked" developers who then lower the quality of their reports to save time and catch up.

As already mentioned, for BSZKB#2, unfortunately, we found no significant correlations for any of the measures.

11.3 Threats to Validity

Generalizability of Results. Software engineering tools and processes vary in different projects and, therefore, our findings based on the projects investigated may not generalize to other projects. However, we analyzed often-used and well-known OSS projects. Therefore, it is reasonable to conclude that the selected projects are no exception in OSS engineering. On the other hand, we were only able to analyze two CSS projects, both provided by the Zurich Cantonal Bank. Therefore, we acknowledge threats in generalizing these results to other CSS projects.

Definition of Software Quality. Our choice of number of bugs as the only indicator of software product quality gives rise to internal

and/or construct validity issues. In particular, we counted bugs when they were reported on the BTS. The reported bugs were introduced a different times before they were found and reported. Due this variable amount of time between bug introducing commit and bug report, we may have time shifting threats in our correlation calculations.

Missing Information. Regarding only bugs listed in the BTS, further threats may arise since, to our knowledge and as shown in Chapter 8, some of the projects (e.g., APACHE) investigated make use of complementary systems to report and discuss bugs (e.g., an email discussion system). Therefore, we may only have considered part of the bug information.

Committers and Developers. We evaluated the number of developers based on the version control log file. In other words, a developer equals to a committer. This may cause problems, since not all OSS developers are allowed to commit and therefore are not counted in our evaluation.

Correlation and Causality. It is important to note that all our explorations are based on correlations. Hence, we cannot make definitive statements on causality.

11.4 Concluding Discussion

In the previous chapter, we analyzed the interplay of data quality measures and bug fixing performance. We extended this work and calculated Kendall tau rank correlations between our data quality measures and number of bugs as a product quality measure. In addition, we extended the calculations with time-shifted correlations to analyze the data for time-shifting effects.

In particular, we showed that product quality—measured by number of bugs reported—is affected by a few of the process data quality measures. Again, the proportion of empty messages leads to curious findings in ECLIPSE. Although we were not able to find strong evidence for correlations between process data quality and software quality, there are tendencies which may have the potential to prompt

practitioners to increase the quality of their software process and its associated data quality.

In the next chapter, we analyze the last aspect of Research Question 4. Specifically, we analyze laws and regulations and search for requirements on accurate data quality in software engineering. Such findings, again, may have the potential to prompt at least CSS practitioners to ensure better data quality in the future since they have to do so by law.

12

When Accurate Data Quality is Required by Laws and Regulations

The quality of software engineering is highly questionable in light of the results presented in the previous sections. Program code changes, unfortunately, are not always properly justified by a rationale. We have also shown that such quality issues lead to bias in datasets and that empirical software engineering applications, such as the award-winning bug prediction algorithm BUGCACHE, are affected by bias. But the effects of these quality issues are not limited to research result; data quality issues also influence the performance of bug fixing. We found tendencies that poor data quality also influences the software product quality. We hope that these findings may prompt software engineering practitioners to enhance their software engineering process data quality in the future.

Although the enforcement of good process data quality by laws, regulations, standards, guidelines, or best practice information technology management frameworks—without obvious and beneficial effect—are mostly of weak motivation, we discuss in this section our last hypothesis:

HYPOTHESIS 4.3: *Laws and regulations nowadays require accurate data quality in software engineering (e.g, justification and traceability of all program code changes).*

We theorize that current laws such as the Sarbanes-Oxley Act [United States Code, 2002] and information technology management frameworks such as the Control Objectives for Information and Related Technology (COBIT) framework [IT Governance Institute, 2007] require the traceability of changes in information systems and data processing, which has an influence on software engineering. Although these specifications and guidelines may not explicitly require accurate data quality (including consistent traceability of program code changes), we see strong tendencies that future principles will do so (especially for certain industries such as banking or insurance business). Thus, at least in well-regulated industries, companies will no longer have a choice, but will have to ensure good data quality, which includes the traceability and documentation of all software changes in any stage of software engineering processes (maybe restricted to CSS). In this section we therefore discuss the most relevant frameworks, standards, guidelines, laws, and regulations that may require accurate data quality in software engineering.

12.1 Information Technology Frameworks

Information technology (IT) frameworks usually define best-practice standards for managing the delivery of cost-effective IT services. Nowadays, companies have the choice of more than 20 different frameworks and standards including: TickIT[1], IT Service CMM[2],

[1] http://www.tickit.org/
[2] http://www.itservicecmm.com/

IT Balanced Scorecard[3], ISO/IEC 27001[4], Six Sigma[5], AS 8015-2005[6] and ISO/IES ISO38500[7], ISO/IEC 20000[8], eSCM-SP[9], COBIT[10], M_o_R[11], BiSL[12], ISPL[13], ITIL[14], ASL[15], MSP[16], PRINCE2[17], PMBOK[18], eTOM[19], and OPM3[20]. However, some frameworks only cover specific aspects of IT, such as security, risk management, procurement, etc., whereas others such as COBIT cover all aspects of IT.

We omit a full discussion of all these frameworks but focus on three often-used frameworks, discussed in the following sub-sections.

12.1.1 Information Technology Infrastructure Library (ITIL)

The Information Technology Infrastructure Library (ITIL) is a set of concepts and best-practices for IT service management, IT development, as well as IT operations [APM Group Ltd, 2010]. The library contains broad and publicly available professional documentation on how to plan, deliver, and support IT service features. ITIL gives detailed descriptions of a number of important IT practices and provides

[3]For example, `http://www.balancedscorecard.org/`
[4]`http://www.iso.org/`
[5]For example, `http://www.asq.org/learn-about-quality/six-sigma/overview/overview.html`
[6]`http://www.ramin.com.au/itgovernance/as8015.html`
[7]`http://www.iso.org/`
[8]`http://www.iso.org/`
[9]`http://www.itsqc.org/`
[10]`http://www.isaca.org/`
[11]`http://www.mor-officialsite.com/`
[12]`http://www.aslbislfoundation.org/`
[13]`http://projekte.fast.de/ISPL/`
[14]`http://www.itil-officialsite.com/`
[15]`http://www.aslbislfoundation.org/`
[16]`http://www.msp-officialsite.com/`
[17]`http://www.prince-officialsite.com/`
[18]`http://www.pmi.org/`
[19]`http://www.tmforum.org/`
[20]`http://www.pmi.org/`

comprehensive checklists, tasks, and procedures that any IT organization can tailor to its needs. In May 2007, the latest version, ITIL v3, was published and comprises five volumes: ITIL Service Strategy, ITIL Service Design, ITIL Service Transition, ITIL Service Operation, and ITIL Continual Service Improvement.

ITIL does not explicitly require traceability of program code changes but defines several requirements for change management as part of ITIL Service Transition. Part of these change management requirements are authorization as well as traceability of changes. Note that ITIL only requires change management for configuration items, and these items may vary widely in complexity, ranging from an entire service or system to a single software or hardware component. As a result, traceability of single program code changes is not explicitly required by ITIL.

12.1.2 Capability Maturity Model Integration (CMMI)

The Capability Maturity Model Integration (CMMI) is a process improvement approach that helps organizations improve their performance. CMMI can be used to guide process improvement across a project, a division, or an entire organization. According to CMMI [CMMI Product Team, 2006], it helps "integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes". In contrast to ITIL, CMMI is less focused on service and infrastructure management and more on product development. The latest version of CMMI—CMMI for Development, Version 1.2—contains 22 process areas that describe the aspects of product development that are to be covered by organizational processes.

Whereas ITIL does not explicitly require traceability, in CMMI there are specific requirements related to the following process areas:

- "The requirements are allocated to product functions and product components including objects, people, and processes. The traceability of requirements to functions, objects, tests, issues, or other entities is documented." (see "SG 2 Develop Product Requirements" [CMMI Product Team, 2006])
- "Requirements traceability can also cover the relationships to other entities such as intermediate and final work products, changes in design documentation, and test plans. The traceability can cover horizontal relationships, such as across interfaces, as well as vertical relationships. Traceability is particularly needed in conducting the impact assessment of requirements changes on the project's activities and work products." (see "SP 1.4 Maintain Bidirectional Traceability of Requirements" [CMMI Product Team, 2006])

12.1.3 Control Objectives for Information and Related Technology (COBIT)

The Control Objectives for Information and related Technology (COBIT) is a framework with a set of best practices for IT management [IT Governance Institute, 2007]. COBIT provides managers, auditors, and IT users with a set of generally accepted measures, indicators, processes, and best practices to assist them in maximizing the benefits derived through the use of information technology and developing appropriate IT governance and control in a company. The framework in its version 4.1 defines 34 high-level processes that cover 210 control objectives categorized in four domains:

1. Plan and Organize
2. Acquire and Implement
3. Deliver and Support
4. Monitor and Evaluate

As with ITIL, COBIT does not explicitly require traceability of program code changes but discusses several related requirements in the domains "Acquire and Implement" and "Deliver and Support". Specifically, the following control objectives may have an impact:

- AI2.10 Application Software Maintenance
- AI6.1 Change Standards and Procedures
- AI6.3 Emergency Changes
- AI6.4 Change Status Tracking and Reporting
- AI7.8 Promotion to Production
- DS9.1 Configuration repository and baseline

12.2 Information Security Standards and Guidelines

Whereas IT management frameworks fulfill IT governance aspects, information security standards and guidelines are focused on risk management and usually share the common goals of protecting the confidentiality (ensuring that information is accessible only to those authorized to have access), integrity (safeguarding the accuracy and completeness of information and processing methods), and availability (ensuring that authorized users have access to information and associated assets when required) of information. Again, we focus on a representative subset.

12.2.1 ISO/IEC 27002

The international standard ISO/IEC 27002, known as the code of practice for information security management, provides best practice recommendations on information security management for use by those who are responsible for initiating, implementing, or maintaining Information Security Management Systems (ISMS) [ISO/IEC, 2005a]. The standard comprises twelve main sections and within each section

information security controls and their objectives are specified and outlined. The information security controls are generally regarded as best practice means of achieving those objectives. Implementation guidance is provided for each of the controls.

Since the ISO/IEC 27002 focuses on information security management, it is not surprising that there are no explicit requirements for traceability of program code changes. Nonetheless, the standard requires special care in changing the configuration in sections "10 – Communications and operations management" and "12 – Information systems acquisition, development and maintenance" as follows [ISO/IEC, 2005a]:

10.1 Operational Procedures and Responsibilities
IT operating responsibilities and procedures should be documented. Changes to IT facilities and systems should be controlled. Duties should be segregated between different people where relevant (e.g., access to development and operational systems should be segregated).

12.5 Security in Development and Support Processes
Application system managers should be responsible for controlling access to [development] project and support environments. Formal change control processes should be applied, including technical reviews. Packaged applications should ideally not be modified. Checks should be made for information leakage for example via covert channels and Trojans if these are a concern. A number of supervisory and monitoring controls are outlined for outsourced development.

12.2.2 BSI IT-Grundschutz Catalogues

The IT-Grundschutz Catalogues is a set of standards in information security provided by the German Federal Office for Information Security (BSI). The aim of IT-Grundschutz is to achieve an appropriate level of security for all types of information of an organization. IT-Grundschutz uses a holistic approach to this process. Through proper application of well-proven technical, organizational, personnel, and infrastructural safeguards, a security level is reached that is suitable and adequate to protect business-related information with normal pro-

tection requirements. In many areas, IT-Grundschutz even provides advice for IT systems and applications requiring a high level of protection [Federal Office for Information Security (BSI), 2005].

Although the BSI IT-Grundschutz Catalogues in its current version is a "book" of 2 922 pages, traceability of program code changes and change management are discussed only marginally.

Module "B 1.9 Hardware and software management" (part of the "Generic Aspects of IT security" layer) formulates the following requirements:

"The purpose of a change management system is to subject revisions to current configurations to a formal documentation and approval process (see S 2.221 Change management). This process includes evaluating aspects for which security is a critical issue as well as executing the changes using the dual-control principle and updating revisions. Another factor is the permitted use of approved components only, as otherwise operations cannot be monitored."

Following the pointer to change management, there is also the following safeguard guidelines for changes:

"There should therefore be guidelines for implementing changes to IT components, software or configuration data. All changes to IT components, software or configuration data should be planned, tested, approved and documented."

The last point is of main interest, since the guidelines explicitly require that all changes to IT components should be documented. Admittedly, there are no requirements for justification and traceability of program code changes, but instead for changes to the configuration in terms of software or hardware components.

12.3 Laws and Regulations

In the previous two sections we discussed IT management frameworks as well as commonly used information security standards and guidelines. Please note that the use of such frameworks, standards, or guidelines in (software) companies/projects is not enforced, although laws and regulations encourage companies to do so (e.g., SOX recommends the adoption of COBIT or COSO). In the next two sub-sections we discuss laws and regulations that may impact the discussion of enforcing accurate data quality in software engineering.

12.3.1 Sarbanes-Oxley Act (SOX)

The Sarbanes-Oxley Act (SOX) is a United States federal law enacted on July 30, 2002 as a reaction to a number of major corporate and accounting scandals with the goal of more reliable financial reporting, greater transparency, and accountability [United States Code, 2002]. SOX was invented for U.S. public company boards, management, and public accounting firms and does not apply to privately held companies. But SOX applies not only to U.S. companies but all companies whose national securities exchanges and equity securities are dealt on American stock markets or whose securities are dealt in public offering. Therefore many international companies have to fulfil the requirements of SOX.

SOX contains 11 sections that describe specific mandates and requirements for financial reporting. Related to IT, the most impactful part of SOX is the controversial and cost-intensive Section 404. Specifically, SOX requires companies to establish adequate internal controls over financial reporting, which results in an increased focus on IT controls, as these support financial processing and therefore fall into the scope of internal control. The related IT control objectives refer to the confidentiality, integrity, and availability of data and the overall management of the IT function of the company which, indeed, enables the requirement of IT management frameworks as well as information security standards.

12.3.2 Revised International Capital Framework (BASEL II)

In 1988 a set of minimal capital requirements for banks (called Basel I) was introduced by central bankers from around the world. Basel I primarily focused on credit risk management and is now widely viewed as outdated, since a more comprehensive set of guidelines, called Basel II, has since been published.

In contrast to Basel I, the Basel II framework describes a more comprehensive measure and minimum standard for capital adequacy. It seeks to improve on the existing rules by aligning regulatory capital requirements more closely to the underlying risks that banks face. Basel II uses a "three pillars" concept to promote greater stability in the financial system: (1) minimum capital requirements, (2) supervisory review, and (3) market discipline.

Regarding IT, the first pillar, which contains operational risk management requirements, is of major interest. An operational risk is a risk arising from the execution of a company's business functions and, therefore, focuses on the risks arising from the people, systems, and processes through which a company operates. In addition, external events are part of operational risk. Since IT has grown in its business importance in recent years, operational risks such as lack of availability of IT systems or internal fraud with IT systems have to be managed. This increases the need to use IT management frameworks and to comply with information security standards and guidelines.

Comparable to SOX, Basel II does not enforce any IT guidelines directly, but requires operational risk management where IT plays a major role. Hence, unjustified and untraceable changes to IT systems and software systems are a major operational risk and have to be managed, for example, by enforcing traceability and justification of all program (code) changes.

12.4 Concluding Discussion

In this section we discussed several frameworks, standards, guidelines, and laws that may require accurate data quality in software engineering as hypothesized. Not surprisingly, we did not find any explicit requirements for accurate data quality in software engineering, since most of these principles do not focus on software engineering and therefore have only high-level requirements for this field.

Whereas CMMI and to some extent the BSI IT-Grundschutz Catalogues explicitly require documentation, change management, as well as traceability of all changes to IT and software systems, other principles' requirements are of an implicit nature. Considering operational risk management, unjustified and untraceable changes to program code and configuration items may lead to serious problems in software systems and their business services. Therefore, at least in well-regulated industries such as banking, operational risk management (which is required by Basel II) should require justification and traceability of all program code and configuration changes including a properly adopted access control system to limit the number of people who are allowed to perform such changes. Regarding commonly adopted information security standards and guidelines, we strongly believe that integrity of data (as required by these standards and guidelines) is not limited to business data but also to data in IT development such as program code, bug reports, and change logs. Future extensions of principles discussed in this section as well as newly developed principles, therefore, may contribute more explicit control requirements for IT development, possibly restricted to CSS projects.

Summarizing, we were able to support Hypothesis 4.3 partially. Regarding Research Question 4 however, principles are mostly of weak motivation. Therefore, frameworks, standards, guidelines, and laws may require better data quality but unless software engineering practitioners can profit from such data quality enhancement, we have no guarantee of a better support of empirical software engineering by software engineering tools either through higher data quality or better export and data conversion support.

Part VI

Summarizing Discussion, Limitations, Future Work, and Conclusions

13

Summarizing Discussion

Software systems today play an important and central role in business as well as private life and are embedded in almost every electronic device. Usually, these software systems are produced by human software engineering experts based on user or business needs and requirements. Unfortunately, humans are imperfect and software systems are usually complex. Hence, software systems may contain bugs and therefore sometimes act not as specified or desired. Most commercial software companies therefore spend much money and effort uncovering all bugs in a software system before it becomes available to the public. But, the increasing software complexity and constraints in time and money do not allow a 100% testing of a software system, software testing rarely uncovers every bug, and "every program has at least one more bug" (Lubarsky's Law of Cybernetic Entomology). Open source software (OSS) projects, in contrast to commercial projects, often pass a testing of software releases by professional testing engineers, but release alpha and beta versions and let users perform testing and reporting/documenting uncovered bugs.

The rising complexity and size of software systems not only constrains a full testing but also has an effect on processes such as the development of software systems, which mostly need attention of more

than one single developer. Hence, efficient handling of bug repair and concurrent development itself increases needs for software tool support. Therefore—as software engineering grew in importance in the last decades—more and more software engineering and testing tools that support practitioners in engineering and maintaining a software system became available. Luckily, these tools (e.g., bug tracking systems, version control systems, integrated development environments) not only support the users and practitioners in their work but also store information about the software engineering and maintenance processes—i.e., bug fixing activities and program changes—and are, therefore, a valuable source of information on the history and evolution of a software system.

Software engineers make use of such information, for example, to define new test cases based on bug reports or to try to understand the history of a software system during bug fixing and refactoring tasks. During the past few years, software engineering process data have also gained popularity among empirical software engineering researchers that use such data to analyze software engineering and develop approaches as well as algorithms. For instance, they evaluate the performance of bug fixing processes, predict the number and locale of bugs in future releases, or analyze when a bug was introduced. Due to the rising popularity of empirical software engineering, many workshops, conferences, and journals address software engineering research and encourage researchers to submit publications in this research area. Though there are complaints about data quality in software engineering datasets, empirical software engineering research leaders believe in an enormous potential of mining software archives [Godfrey et al., 2009].

Unfortunately, software engineering process tools mostly do not provide integrated and well-prepared process data for research purposes. Therefore, researchers have to develop procedures to gather, prepare, and integrate (i.e., inter-link) software engineering process data. Researchers have to make process assumptions and develop tools, heuristics, and algorithms to prepare software engineering datasets to enable empirical software engineering research. However,

the quality of software engineering process data stored in process tools is questionable and data gathering/preparation techniques developed by researchers are mostly inexact and are based on heuristics. Therefore, the quality of the extracted data (i.e., its completeness and correctness) is unknown and enables the assumption that such data is affected by quality issues (e.g., missing information). Nonetheless, empirical software engineering research is a very popular field and many researchers have presented valuable and promising research results that mostly rely on open source software (OSS).

Specifically, researchers extract software engineering process data from version control systems (VCSs) and bug tracking systems (BTSs) (Figure 13.1-2a and 13.1-2b) and use data preparation techniques (Figure 13.1-3) to create software engineering datasets (Figure 13.1-4). Later, they use these datasets to develop promising research results (Figure 13.1-5).

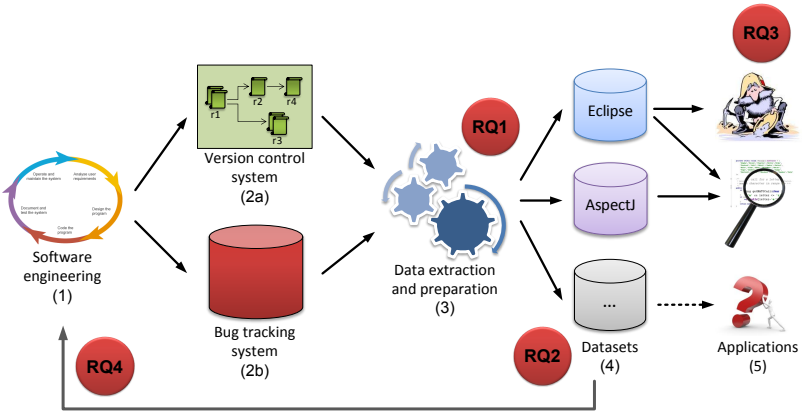


Figure 13.1: Embedding of our research questions among empirical software engineering activities

Impressed by the possibilities and results in the field of empirical software engineering on the one hand and possible data quality issues on the other, we analyzed recent publications and research results. We found that empirical software engineering researchers (see discussion in Section 2) make use of software engineering process data for promising research mostly without addressing possible effects on research results by data quality issues (see [Liebchen and Shepperd, 2008]).

We identified several questions during the course of empirical software engineering activities (Figure 13.1). First, we found that commonly used data extraction and preparation techniques are inexact and produce incomplete datasets. Therefore, we defined our first research question.

RQ 1: *How can we counter the known issues in preparing and linking software engineering process data?*

According to previous work (Chapter 2) and our own findings, the data quality of software engineering datasets is questionable. In addition, many software engineering studies make use of OSS project data (due its availability) but claim considerable financial advantages for closed source software projects (i.e., commercial) projects, despite the different processes and testing approaches used by them. Hence, we tried to qualify and characterize software engineering datasets and therefore asked our second research question.

RQ 2: *How can we qualify and characterize software engineering process data for evaluation and comparison across projects?*

However, evaluating software engineering process data quality and characteristics and reporting poor data quality is of no importance if there is no effect on research results. With our next research question, therefore, we analyzed why empirical software engineering researchers should care about possible data quality issues.

RQ 3: *Why should empirical software engineering researchers care about data quality in software engineering?*

How can we counter possible data quality issues and ensure better data quality in the future? To address this question, we remind the reader that software engineering process data is stored by software engineering tools used by practitioners. A major challenge we address in our first research question is the lack of data integration between VCSs and BTSs. Researchers have to rely on linking information provided by developers in VCS log messages, which is often of poor quality. With our last research question we analyze why practitioners should care about data quality and so should spend more effort and money to ensure better data quality in the future.

RQ 4: *Why should software engineering practitioners care about data quality in software engineering?*

To analyze our research questions and achieve deeper knowledge about data quality issues and possible effects, we used several software engineering process datasets in our work. In addition, we discussed current challenges of empirical software engineering research in more detail. We first introduced commonly used software engineering processes and the tools used. These tools usually store valuable software engineering process data (Chapter 3). At the same time, we also discussed the problem of missing integration (i.e., linking) between VCSs and BTSs. We then introduced six popular and often-used OSS as well as two CSS projects (Chapter 4):

- APACHE HTTP WEB SERVER (OSS)
- ECLIPSE IDE (OSS)
- GNOME Desktop Suite (OSS)
- NETBEANS IDE (OSS)
- OPENOFFICE (OSS)
- MOZILLA (OSS)
- BSZKB#1 – Banking System 1 (CSS)
- BSZKB#2 – Banking System 2 (CSS)

All these projects have a long-term project history with many users or systems involved and play an important role in their field. With our choice of the two CSS projects provided by the Zurich Cantonal

Bank, we believe to have achieved at least a small insight into commercial practices. In addition to the datasets prepared by ourselves, we used, for comparison and validation reasons, the ECLIPSE$_Z$ dataset provided by Zimmermann *et al.* [Dallmeier and Zimmermann, 2007; Zimmermann et al., 2007], which is well-documented, has been widely used in research (e.g., [Moser et al., 2008; Čubranić et al., 2005; Zimmermann et al., 2007]) as well as in the Mining Software Repositories Conferences' mining challenges in the years 2007[1] and 2008[2], and is available for download on the Internet[3].

Using these datasets, we were able to explore our four research questions and test several hypotheses. In the following sections, we provide a contextual discussion and present the results/findings of the hypotheses analyzed and tested for each of the research questions discussed above.

13.1 How can We Counter the Known Issues in Preparing and Linking Software Engineering Process Data?

Nowadays, almost every medium- to large-scale software project makes use of a VCS to track all software changes and allow concurrent development activities by multiple developers. CVS and SVN are both very popular VCSs but use different methods of versioning the data, which results in, depending on the research goals, additional effort by researchers. In addition, BTSs such as BUGZILLA, ISSUEZILLA, QUALITY CENTER, or JIRA are often used in software projects to manage and track software bugs. Unfortunately, VCSs and BTSs are mostly not integrated and, therefore, the process data (e.g., VCS log messages and BTS bug reports) is not linked. Project

[1] http://msr.uwaterloo.ca/msr2007/challenge/
[2] http://msr.uwaterloo.ca/msr2008/challenge/
[3] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

solutions such as IBM JAZZ[4] which combine, for example, the functionality of bug tracking and version control solutions, would provide integrated data but are not very popular and only a few projects use them.

In order to overcome these issues, empirical software engineering researchers have developed several procedures, heuristics, and tools to create datasets for empirical software engineering datasets. To do so, they used data extraction techniques to download all relevant information from the VCS and BTS, developed approaches to reconstruct the original commits from CVS log files, and used heuristics to integrate (i.e., link) the data. Such integration usually relies on VCS log messages with bug report numbers referenced (see, e.g., [Fischer et al., 2003b; Śliwerski et al., 2005; Čubranić and Murphy, 2003; Zimmermann et al., 2007]). Conscious developers add this information during the check-in process of new software versions to a VCS. Unfortunately, linking approaches are inexact and datasets may contain missing and wrong information. In addition, many approaches only consider part of the information provided by software engineering tools.

In order to overcome the known issues in data preparation and linking, we presented in Chapter 5 a step-by-step procedure to gather, convert, and link software engineering process data and introduced an extension of the original linking approach provided by Fischer *et al.* [Fischer et al., 2003b].

Specifically, we analyzed and discussed three hypotheses related to our first research question:

HYPOTHESIS 1.1: *Our procedure addresses the known issues in preparing, converting, and linking software engineering process data and enhances existing algorithms.*

HYPOTHESIS 1.2: *Our extended algorithm produces datasets with a higher linking ratio as well as data quality than those previously presented.*

[4] `http://jazz.net/`

HYPOTHESIS 1.3: *Our data preparation technique produces datasets with a more complete picture of software engineering process data than those previously presented.*

Compared to previously presented data extraction approaches and datasets, we extracted all projects' BTS information including the bug activity information for every bug. The extraction and preparation of all data stored in software engineering tools allowed us to analyze the data quality in more detail. In addition, we achieved a more complete picture of these data (supporting Hypothesis 1.3), allowing us to analyze the data for incognito bugs (Chapter 8) as well as commit feature and bug feature bias (Chapter 9).

We extracted the BTS information by downloading all the relevant files (OSS datasets) or performed a database dump (CSS datasets) and extracted the VCS log files. We then parsed / loaded the data into a relational database and reconstructed the transactions—if necessary—based on heuristics and the use of author, time stamp, and commit message information (as presented in Section 5.3), which is common practice (e.g., [Breu and Zimmermann, 2006; Breu et al., 2006; Zimmermann and Weissgerber, 2004]). However, such transaction reconstruction is inexact and may result in non-complete transactions, for example, if developers upload very large files or have a slow connection to the Internet. Specifically, we merged all CVS log entries with the same commit message and author as well as time delays across commits below five seconds (sliding window approach) to one transaction.

The integration (i.e., linking) of the VCS and BTS information was based on previously presented approaches, but several changes were made to decrease the number of false-negative links (i.e., links that are valid references in the commit log but not recognized by the current approach).

Specifically, we relaxed the pattern-matching to find more potential mentions of bug reports in commit log messages and then verified these references more carefully in three steps (Steps 2–4):

1. We scan through the commit/transaction messages for numbers in a given format, or numbers in combination with a given set of keywords.
2. We exclude all false-positive numbers (e.g., release numbers or calendar dates), which have a defined format.
3. We check if the potential bug number exists in the BTS.
4. We check if the linked bug report is a fixed bug report and whether it has a fixing activity seven days before or seven days after the commit date.

As discussed in Section 5.4, the heuristic used in the last step of our procedure reflects the main difference to other approaches. Using this approach to link software engineering process data, we were able to increase the linking ratio significantly compared to previous approaches, counter known issues in linking, and provide accurate datasets for further research, supporting Hypotheses 1.1 and 1.2.

But does our algorithm not only produce a higher linking ratio but also more false-positive links, and how many links have we left out (false-negative results)? We have manually inspected thousands of randomly selected VCS log messages across all projects investigated and found only a few false-positives such as year dates or release numbers which were recognized and verified as valid bug report. Checking for false-negative links, on the other hand, needs more attention. Our algorithm defines a valid time-period of ± 7 days between commit date and status change to fixed in the bug report. Of course, not all valid links will be caught by this heuristic and therefore false-negatives will appear. Being more lax and defining a less restrictive time constraint, on the other hand, may result in more false-positive results. Therefore, it is a trade-off between more false-negative and fewer false-positive results. Nonetheless, the higher the time difference between commit and bug report status change, the lower the likelihood of a valid link. Again, we checked thousands of VCS log

messages for false-negative results across all projects and found only a few. Table 13.1 lists the false-negative and false-positive rates as well as the corresponding confidence intervals of our manual inspection for all datasets considered.

In addition to our own manual inspection of all datasets, we engaged Dr. Justin Erenkrantz, an APACHE expert developer, to manually inspect all the links of a six-week period in the APACHE HTTP WEB SERVER project dataset. Surprisingly, he found no false-positive links and only three false-negatives (again, see Table 13.1). Note that these false-negatives did not satisfy our heuristics for valid links and our heuristics are inexact as well. Hence we have a trade-off between false-positive and false-negative results but decided to calibrate the heuristics in such a way. We concluded that the extremely low levels of observed error in our and Justin's manual examination did not pose a threat, and so we assumed that our linking algorithm finds virtually all the commit log messages which the developers flagged as fixing specific bugs in all projects investigated (Hypothesis 1.2).

During our detailed manual examination of our datasets and the data stored in software engineering tools we found, unfortunately, that only a fraction of fixed bugs are mentioned in VCS log messages. Therefore, linking algorithms—even enhanced ones—are able to link only a fraction of fixed bug reports. However, this is not affected by poor linking algorithms but rather by missing information in VCS log messages. Therefore, we have to conclude that not only datasets prepared from raw data may be plagued by quality issues but also the raw data itself. Such data quality issues may have an effect on research results and may affect software engineering processes as well as product quality. With Research Question 2, therefore, we address the question of how we are able to qualify and evaluate software engineering process data.

Table 13.1: Observed linking error in datasets (extract of Table 5.4)

	False-negative rate (95% Confidence interval)	False-positive rate (95% Confidence interval)
APACHE complete	0.000835 [0.000464 , 0.001468]	0.000000 [0.000000 , 0.004893]
APACHE evaluation sample	0.006466 [0.001671 , 0.020408]	0.000000 [0.000000 , 0.145616]
ECLIPSE	0.004000 [0.001627 , 0.009153]	0.000200 [0.000081 , 0.000459]
GNOME	0.000153 [0.000095 , 0.000243]	0.000386 [0.000124 , 0.001060]
NETBEANS	0.000121 [0.000059 , 0.000239]	0.000053 [0.000003 , 0.000346]
OPENOFFICE	0.000343 [0.000238 , 0.000490]	0.00000 [0.000000 , 0.000359]
MOZILLA	0.000365 [0.000198 , 0.000657]	0.000113 [0.000020 , 0.000455]
BSZKB#1	0.000364 [0.000116 , 0.001000]	0.000000 [0.000000 , 0.002554]
BSZKB#2	0.000137 [0.000035 , 0.000436]	0.000000 [0.000000 , 0.006728]

13.2 How can We Qualify and Characterize Software Engineering Process Data for Evaluation and Comparison across Projects?

With the knowledge of how to prepare software engineering process data for empirical software engineering research, we now address the question of how we are able to qualify and characterize such data. We

have two goals asking this question: First, we are looking for the ability to evaluate data quality in software engineering datasets (e.g., the linking ratio), allowing a comparison across projects as well as reporting the degree of quality. Second, we are looking for the ability to evaluate data characteristics, allowing a comparison of tool-usage and process characteristics across projects, where the comparison of OSS and CSS projects is of particular interest.

Specifically, we analyze the following hypotheses:

HYPOTHESIS 2.1: *Our framework of data measures can evaluate and compare the data characteristics and quality across several software projects.*

HYPOTHESIS 2.2: *Software engineering datasets are plagued by data quality issues such as missing information.*

HYPOTHESIS 2.3: *Software engineering datasets vary in their characteristics across projects, especially between open source and closed source software projects.*

We introduced a framework of twenty data quality and characteristics measures (Chapter 6). We then evaluated the data quality of all our datasets and found, not surprisingly, that all projects—both OSS and CSS projects—have data quality issues (Chapter 7). In all projects, the linking ratio (a quality measure) between VCS log messages and (fixed) bug reports is on a poor level of below 55%, supporting Hypothesis 2.2. In the worst case (OPENOFFICE), only 7.43% of all fixed bug reports are properly mentioned in VCS log messages. This raises the question of whether the subset of linked bug reports is a representative sample of all fixed bug reports. If not, such datasets may be biased and research results may be affected. In Chapter 9, therefore, we analyzed the datasets for bias and possible effects on research results (see discussion below).

In addition, we used our data characteristics measures to evaluate the data characteristics across all projects investigated and found,

again not surprisingly, vast differences across all projects, especially between OSS and CSS projects, supporting Hypothesis 2.3. This puts at risk the generalization of research results. Although we have not further analyzed these issues, we strongly believe that research results may not generalize and cannot be re-used across projects.

Summarizing, our framework of data quality and characteristics measures allows a fast and comprehensive view on software engineering datasets, supporting Hypothesis 2.1. We encourage empirical software engineering researchers to make use of these measures and report the evaluated values in future empirical studies and publications, leading to at least two beneficial effects: First, the reported measures values show what kind of software engineering data was used (data characteristics). Second, the values show the quality of the data used and are, therefore, a good indicator of the generalizability of results and possible threats to validity. Analyzing our datasets, we have shown that data characteristics vary across OSS and CSS projects. Empty VCS log messages, for instance, seem to appear only in the ECLIPSE project. All other projects have a ratio of empty messages far below ECLIPSE that of. In addition, we have shown that software engineering datasets are plagued by quality issues such that even with an enhanced linking algorithm only a fraction of fixed bug reports are mentioned in VCS messages (in the best case 50%).

However, evaluating and reporting data quality issues can only be the first part of the story. If there is no effect on (i) results of empirical software engineering studies or (ii) the work of practitioners, we can ignore these issues. With Research Questions 3 and 4, therefore, we address the questions why we should care about these issues.

13.3 Why Should Empirical Software Engineering Researchers Care about Data Quality in Software Engineering?

As discussed in the previous sections, empirical software engineering researchers make use of software engineering process data for promising research. First, we introduced an extension of previously presented approaches for preparing and linking software engineering process datasets. Second, using our framework of data quality and characteristics measures, we uncovered numerous data quality issues and differences in data characteristics across projects investigated. These findings may raise major threats in research results which are based on such data, although most publications and studies do not even (properly) address quality impacts and possible threats to results (see [Liebchen and Shepperd, 2008]).

But do these findings really have an effect on research results? To address this question and others, we analyzed three hypotheses:

HYPOTHESIS 3.1: *Process assumptions made by empirical software engineering researchers may be wrong.*

HYPOTHESIS 3.2: *Software engineering datasets are plagued by bias due to a lack of complete linking.*

HYPOTHESIS 3.3: *Bias in software engineering datasets has an effect on empirical software engineering results.*

Ideally, all bug fixing commits are linked to fixed bug reports. However, as already discussed, only a fraction of fixed bugs have links to bug fixing commits. This raises the possibility of two types of bias in data: *Bug feature bias*, where only certain types of bugs are linked, or *commit feature bias*, whereby only certain types of bug fixing commits

are linked. First, we analyzed the datasets and found, unsurprisingly, that our datasets are affected by bug feature bias: Less severe bug reports, for instance, are more likely to be linked (Hypothesis 3.2). But are research results affected by biased data? To uncover the effects of bias on research results, we implemented BUGCACHE, an award-winning bug prediction algorithm. We the analyzed the impact of biased datasets on results and showed that bug feature bias affects the performance of BUGCACHE, supporting Hypothesis 3.3.

Unfortunately, we were not able to analyze the datasets for commit feature bias without having the ground truth for at least a subset of commit data. Therefore, we used the services of Dr. Justin Erenkrantz, an APACHE core developer, to find at the ground truth in a subset of the original APACHE HTTP WEB SERVER project dataset. Based on this verified dataset we were able to verify our data gathering, processing, and linking approach (as already discussed). In addition, Justin provided us with important information on the practices used in the APACHE HTTP WEB SERVER project. We learned that things are even worse than previously thought and process assumptions made by researchers may be wrong: The most important bugs often never show up in the APACHE BTS but are instead discussed on the APACHE email discussion system (Chapter 8). This supports Hypothesis 3.1 and again raises questions about the reliability of research studies, at least for those that used APACHE project data for their experiments. Given that the practices in APACHE are no exception in OSS development, for other projects similar issues may arise. In addition, using the verified dataset, we were able to analyze the effect of commit feature bias on research results. Here we showed that we have differing linking behavior between developers and that there are tendencies with regard to the number of bugs committed and the ownership of the file with respect to the author of the committed fix. Again, we tested whether BUGCACHE is affected by this kind of bias and showed that BUGCACHE has strong tendencies to miss predictions if it is not trained on ground truth data, again supporting Hypothesis 3.3 (Chapter 9).

With our work we therefore have shown that empirical software

engineering researchers should care about software engineering process data quality and characteristics and why. First, our analyses showed that the sample of linked bug reports in all datasets investigated is biased and that bug prediction algorithms such as BUGCACHE may be affected by biased data. Second, we have shown that things are even worse than thought, because tools are used differently across projects which means that, for instance, in APACHE the most important bugs never show up in the official BTS against process assumptions commonly made by researchers. Such bugs, including security relevant bugs, are likely being discussed by core developers and expert users on the email discussion system, which is used, at the same time, to propose bug fixes to reported bugs. Empirical software engineering researchers, therefore, will need to take the whole software development social eco-system (version control systems, bug tracking databases, email discussions, discussion boards, chats, etc.) into account in order to elicit a more complete picture of underlying software engineering processes.

We uncovered several process data quality issues (e.g., bug and commit feature bias) affecting promising empirical software engineering research results. In previous work, we tried to deal with such data quality issues and evaluated solutions, for example, to fix missing linkage. Specifically, we used author, changed files, and date/time information from the BTS and the VCS to learn from known and verified links to uncover unknown links. Unfortunately, such data quality issues are very hard to fix by empirical software engineering researchers and we were not successfully with our work. Therefore, we have to conclude that we are only able to uncover and report these issues for example through the use of data quality measures as discussed in Chapter 6. However, the best way to ensure good data quality is to ensure qualitative data at the source of information by software engineering practitioners and users of software engineering tools. But why should practitioners, for example, bug report authors and developers, spend time, effort, and money to ensure better data quality? With Research Question 4 we address this question and discuss why practitioners should care about data quality and should improve data

quality in the future.

13.4 Why Should Software Engineering Practitioners Care about Data Quality in Software Engineering?

In the previous sections we discussed that software engineering process datasets are plagued by quality issues such as missing links and empty commit messages, leading to bias in these datasets. Based on our experiments we have also shown that such quality issues have a major impact on empirical software engineering research results. Nonetheless, according to research leaders in empirical software engineering [Godfrey et al., 2009], this research field has an enormous potential, such that practitioners in the future should have better tool support in their daily work and profit from knowledge currently still hidden, which may increase work efficiency and allow faster and more qualitative software engineering. Tables 13.2 and 13.3 at the end of this chapter list current research results, which have the tendency to support practitioners in such a way (see Chapter 2 for a discussion). However, empirical software engineering researchers need accurate data quality, otherwise promising research (i) will never be developed into a product or (ii) will be developed but produces wrong results and, therefore, never gains acceptance by customers and practitioners.

Empirical software engineering researchers are, in a limited way, able to deal with data quality issues by implementing heuristics, learning approaches, missing data techniques (which all are inexact) and counter these issues (again see Chapter 2). Still, the easiest and best way to achieve good data quality is to ensure good data quality at its source, rather than trying to fix poor data quality through extensive efforts later.

This raises the question of why practitioners should spend more

time, money, and effort on ensuring better data quality in the future. The answer is that practitioners could profit from future results of empirical software engineering research (as Tables 13.2 and 13.3 show). In addition, we analyzed and discussed the following three hypotheses, which may prompt practitioners to enhance the data quality in software engineering processes and tools:

HYPOTHESIS 4.1: *Poor software engineering data quality influences the bug fixing process and bug fixing activities (i.e., performance of bug fixing).*

HYPOTHESIS 4.2: *Software engineering process quality (evaluated by process data quality) influences product quality (measured by number of bugs).*

HYPOTHESIS 4.3: *Laws and regulations nowadays require accurate data quality in software engineering (e.g, justification and traceability of all program code changes).*

For most practitioners, future benefits of empirical software engineering results may be a pleasing side-effect but not an argument to spend more time and money today. Direct effects on today's work (e.g., product quality or process performance), in contrast, have the tendency to be much stronger arguments for practitioners to ensure accurate process data quality in the future. Therefore, we analyzed such effects with Hypothesis 4.1 and 4.2. We theorized that the better the process data quality, the less effort for maintenance is needed and the fewer the bugs are produced in future releases. In addition, we believe that an accurate documentation of program changes reduce the time which is needed to fix bugs and helps new developers to get more quickly into the work. Specifically, we calculated the data quality and characteristics measures we introduced in Section 6 on weekly data frames. We then calculated Kendall tau rank correlations based on these measures and analyzed possible correlations. To counter time-shifting effects, we also calculated Kendall tau rank

correlations on time-shifted data frames. Analyzing the Kendall tau values, we identified, as hypothesized, several correlations. The proportion of empty messages in ECLIPSE, for instance, correlates with bug report quality. Unfortunately, we were not able to prove causality, but we strongly believe that these correlations indicate (in a limited way) causality. Unfortunately, we were able to support Hypothesis 4.1 only partially—for some of the projects and measures investigated.

In addition, we analyzed Kendall tau correlations between process data quality measures and number of bugs reported. Interestingly, we were not able to find any correlations between the linking ratio (which is on a poor level in all datasets) and number of future bugs. We also analyzed these correlations for time shifting effects and found, again, no significant results. In summary, we were not able to support Hypothesis 4.2 for any of the data quality measures except for the empty messages measure in ECLIPSE. Nonetheless, regarding the problem of missing justification and traceability of program code changes, we strongly believe that such data quality issues have a negative effect on work efficiency because developers may need more time during bug fixing activities (e.g., they need more time to locate the bug introducing commit).

With Hypothesis 4.3 we analyzed if contemporary laws such as the Sarbanes-Oxley Act [United States Code, 2002] and IT management frameworks such as COBIT (Control Objectives for Information and Related Technology) [IT Governance Institute, 2007] require traceability of changes of information systems and data processing. Although there are no explicit requirements for accurate data quality (including consistently traceability of program code changes) in currently enforced rules, regulations, laws, and IT frameworks, we see strong tendencies that future regulations (especially in certain industries such as banking or health care) will do so. In addition, in recent years, companies have started to manage their operational risks, something that is, in some industries, required by laws and regulations such as Basel II [Basel Committee on Banking Supervision, 2006] in the banking industry. As information technology and software systems grow in importance for companies, a failure of these systems is now one of

the most important operational risks which has to be managed. This increases the requirements on software engineering and testing processes as well as justification and traceability of all changes in software systems. Indeed, enforcement by rules, regulations, or managers without obvious and beneficial effect are mostly of weak motivation. Nonetheless, we strongly believe that, at least in well-regulated industries such as banking and insurance business, companies will no longer have a choice, but have to ensure traceability and documentation of changes across the whole software engineering process.

Summarizing, we showed that not only empirical software engineering researchers should care about process data quality but also practitioners. Regarding the beneficial effect of empirical software engineering results on software engineering, practitioners should ask themselves whether they should increase their efforts to ensure better data quality and, therefore, profit from future results and tools (see Tables 13.2 and 13.3). In addition, much effort is nowadays required by researchers to gather, convert, and link software engineering process data for promising research results and applications. Tools with better support to extract, download, and access software engineering data would ease empirical software engineering research and the development of tools that assist practitioners in such analyses. Regarding the hypotheses, unfortunately, we found (almost) no evidence for Hypotheses 4.1 and 4.2. Nonetheless, there are indicators that such correlations can not be fully disregarded. Analyzing Hypothesis 4.3, we found that current rules and regulations do not explicitly require accurate data quality in software engineering but industry-specific rules and laws such as Basel II have the tendency for such requirements. In the face of increased statutory and regulatory standards, we strongly believe that it is only a matter of time before accurate data quality in software engineering is no longer a choice but required and enforced.

These findings have the potential to prompt practitioners to increase the quality of their software engineering processes and the associated data quality. Nonetheless, we encourage researchers to further develop solutions for dealing with poor data quality, even though this is a very difficult task. Even if we enhance the quality of process data

from now, it will take years to get a long-term project data history like we currently have. In addition, further effort should be provided to get more proven datasets to verify results and develop solutions to deal with poor data quality (see Section 14.2).

Table 13.2: Beneficial empirical software engineering results (part 1)

Research field	Research goals, results, and publications
Bug and refactoring prediction	- Predictions of the number and locale of bugs in future software releases which allows the allocation of limited testing resources as efficiently as possible [Askari and Holt, 2006; Bernstein et al., 2007; Catal and Diri, 2009; Ekanayake et al., 2009; Hassan and Holt, 2005; Joshi et al., 2007; Kim et al., 2007; Knab et al., 2006; Nagappan et al., 2006; Neuhaus et al., 2007; Ostrand et al., 2005; Zimmermann et al., 2007] - Detection of bug introducing activities [Aversano et al., 2007; Sahoo et al., 2010; Schröter et al., 2006b; Zimmermann et al., 2004] - Prediction what parts of the program should be refactored [Graves et al., 2000; Ratzinger et al., 2007]
Bug fixing process optimization	- Automatic classification of bug reports (e.g., bug severity or bug category) supporting (semi-)automatic triage and prioritization [Antoniol et al., 2008; Anvik et al., 2006; Gegick et al., 2010; Lamkanfi et al., 2010] - Automatic identification of duplicate bug reports [Bettenburg et al., 2008] - Predictions of how long it will take to fix a bug [Kim and Whitehead, 2006; Panjer, 2007; Weiss et al., 2007] - Enhancements of bug report quality (e.g., verification of bug report quality whilst typing) [Bettenburg et al., 2007a,b; Hooimeijer and Weimer, 2007; Just et al., 2008; Ko et al., 2006; Schugerl et al., 2008] - Automatic bug localization and identification of who originally introduced a given bug, when, and why [Dallmeier and Zimmermann, 2007; Kim et al., 2006a,b; Williams and Hollingsworth, 2004]

Table 13.3: Beneficial empirical software engineering results (part 2)

Research field	Research goals, results, and publications
Software engineering process analysis	- Evaluation of software engineering process characteristics and quality [Bachmann and Bernstein, 2009b; Herraiz et al., 2005] - Verification of effects by software process improvement methodologies on software quality [Ashrafi, 2003; Harter and Slaughter, 2000; Kroeger and Davidson, 2009] - Estimation of future software project costs (e.g., based on software metrics from the past) [Basili et al., 1994]
Software evolution analysis	- Support new developers in understanding the software and its history [Ratzinger et al., 2005; Čubranić and Murphy, 2003; Čubranić et al., 2005] - Visualization of the evolution of software systems, for example, to uncover hidden, shifted, or removed dependencies or better understand the software architecture [D'Ambros et al., 2005; Fischer and Gall, 2006; Fischer et al., 2003a; Lanza, 2003; Pinzger, 2005; Pinzger et al., 2005; Ratzinger et al., 2005]

14

Limitations and Future Work

While we have presented many interesting findings, we also have to highlight some limitations of our work. In addition we present future areas of research uncovered by this thesis.

14.1 Generalization of Results

Software engineering tools and processes vary in different projects and, therefore, our findings and results based on the used projects may not generalize. However, we analyzed six often-used and well-known OSS projects. Therefore, it is reasonable to conclude that the selected projects are no exception in OSS engineering. The two analyzed CSS datasets, on the other hand, were provided by the Zurich Cantonal Bank only. Unfortunately, getting CSS data (even without program code information) is a very time-consuming task and requires much effort. We agree that this limited selection of CSS projects only allows a small insight into commercial software engineering practices of only one commercial company. Therefore, we acknowledge possible generalization problems for other CSS projects. We strongly support an extension of our work to other CSS and OSS projects to get a wider view on data quality and characteristics in software engineering.

14.2 Ground Truth in Software Engineering Datasets

To verify some parts of our work, we engaged the services of an APACHE expert developer to fully annotate and classify a subset of the original APACHE dataset. This may cause two concerns: First, did we choose our time-frame carefully? Second, have we annotated the sub-dataset carefully and really got the ground truth? We analyzed the whole original APACHE dataset based on week-long periods to find a very typical period for our sub-dataset which was as representative as possible in terms of its descriptive statistics. The annotation and classification were performed carefully by a very experienced APACHE expert developer with the support of a self-developed tool. Still, there may be errors. Nonetheless, according to Justin Erenkrantz, the interesting practices of the APACHE developers are by no means exceptional to the selected time period.

Limited by resources of time and money, we were only able to ensure ground truth for one single project, only a limited time-window of six consecutive weeks, and by only one single annotator. Getting the same data annotated by other developers, and checking agreement, would have been better. Therefore, we hope to influence the community and other researchers to seek more ground truth for more software engineering datasets. Granted, such work would entail significant manual labor, but, the resulting verified datasets would undoubtedly help to further verify existing heuristics, tools, and algorithms and allow further empirical software engineering research based on verified datasets. Predictions and pre-selection of data would help to focus the manual work on the most relevant parts of data. Therefore, in future work we should also invest effort in algorithms and learning techniques to enlighten the most relevant data entries in order to use limited resources as efficiently as possible.

14.3 Data Preparation Techniques

The techniques we used for data extraction, conversion, and linking are based on heuristics which are inexact. Even if we have checked our linking approach for false-positive and false-negative results, we may still have errors. Unfortunately, unless we get well-integrated data, we have to use such inexact techniques as other researchers do. Unfortunately, due to missing information in VCS log messages, even with our enhanced linking approach, we were able to link only a fraction of fixed bug reports. As discussed in this thesis, the best way to counter such data quality issues is to produce better data quality in its creation. On the other hand, the long-term project history available in BTSs and VCSs is a very valuable source of information. The data in these systems, unfortunately, is of poor quality. We can certainly try to achieve better data quality in the future, but it will take a long time to get a long-term history with accurate data quality. Therefore, we suggest further analyzing possible techniques and learning approaches to counter the problem of missing information. We tried several ways of doing this, unfortunately without any success.

14.4 Bugs Incognito

As we have shown in the APACHE project, the most relevant bugs never show up in the BTS but are instead discussed on the APACHE email discussion system. Given that the bug reporting practices in APACHE are no exception in OSS software engineering, we strongly recommend taking the whole software engineering social eco-system (including version control systems, bug tracking databases, email discussion systems, discussion boards, chats, etc.) into account in order to elicit a more complete picture of the underlying software engineering processes. This would allow the capturing of bugs and commits which stay incognito outside the BTS.

The data quality and characteristics measures presented in this thesis consider BTS and VCS data only. In future work, therefore, we have

to extend our measures framework to evaluate other sources of information as discussed above.

14.5 Definition and Evaluation of Product Quality

For this thesis, we have chosen the number of bugs as the only indicator of software product quality which may give rise to internal and/or construct validity issues. In future work, we hope to find other empirical measures to define product quality in terms other than only reported bugs. Surveying developers, on the other hand, fits only for current product quality and does not allow a historical view on product quality as we are able to do with number of bugs as the product quality measure.

14.6 Correlations and Causality in Software Engineering Datasets

In Chapter 10 and 11 we calculated Kendall tau rank correlations to test the hypotheses that process data quality influences the bug fixing process and/or product quality. We found tenencies to support these hypotheses but found no evidence in the data to do so. Nonetheless, we believe that process data quality has an effect on process performance as well as product quality, as shown by other researchers (e.g., [Diaz and Sligo, 1997; Harter and Slaughter, 2000; Tajima and Matsubara, 1981]). Therefore, we strongly support further research in analyzing not only correlations between process data quality, process efficiency, and product quality but also causality of these correlations. Such results would have strong tendencies to persuade software engineering practitioners to ensure better data quality and provide tool support for empirical software engineering in the future.

14.7 Influence of New Software Engineering Tools

Nowadays, stand-alone BTSs (e.g., BUGZILLA, ISSUEZILLA, JIRA, and QUALITY CENTER) and VCSs (e.g., CVS and SVN) are widely used in OSS and CSS projects. Unfortunately, as already discussed, these systems are not well-integrated and do not support empirical software engineering research well.

In 2007, Erich Gamma presented a new generation of tool named JAZZ. In contrast to other software engineering tools, JAZZ unites the functionality of several stand-alone tools such as bug tracking systems, version control systems, and integrated development environments (IDEs) into one single software project management solution. One of the basic concepts of the JAZZ solution are tasks. All changes to the program code are planned as tasks where, for example, bug fixes, feature requests, user requirements, and refactoring are tasks in the language of JAZZ. But in contrast to previous tools, all these tasks have to be described and later assigned to a developer. Unless a developer has a task, he can not commit a new version of the program code to the JAZZ solution. Therefore all program changes should be justified by assigned tasks, which is, on first sight, much to the joy of everyone involved in empirical software engineering, because the data should be of good quality. Unfortunately, people tend to act irrationally to enforcements and, if constraints are too rigorous, act against such enforcements and search for ways to by-pass them (e.g., they create a dummy task and assign multiple program changes to this single dummy task). Therefore, it would be of substantial interest to analyze projects which make use of tools such as JAZZ and get an insight into the data quality and characteristics. This would allow a central question to be answered: Do such tools really improve the data quality as supposed?

15

Conclusions

In this thesis, we presented an enhanced step-by-step procedure to prepare software engineering process data for empirical software engineering research. In addition, we introduced data quality and characteristics measures to evaluate the quality of such data. Based on six open source software and two closed source software projects, we calculated these measures and found vast differences in data characteristics across projects as well as data quality issues in all projects. Later we showed why empirical software engineering researchers and practitioners should care about these issues. We showed that software engineering datasets are plagued by commit feature and bug feature bias and that both kinds of bias have an impact on BUGCACHE, a famous bug prediction algorithm. Engaging an APACHE expert developer, we also uncovered the bug reporting practices in the APACHE HTTP WEB SERVER projects, and showed that, even worse, the most important bugs never show up in the APACHE bug tracking database but are instead discussed on the APACHE email discussion system. Hence, practitioners as well as researchers should care about data quality issues, as we have shown that process data quality may influence the product quality. In addition, researchers are not able to develop tools which may help developers in the future. We also explored laws and regulations and their impact on software engineering and showed that there are no explicit but there are implicit requirements for traceability and justification of all program changes.

In summary, with our work we have shown that empirical software engineering researchers and practitioners should both care about data quality, even though their motivations are different. In future work, approaches to (i) counter and deal with poor data quality in existing datasets and (ii) improve the quality of future software engineering process data, for example, by using tools such as JAZZ, should be analyzed in more detail. As a first step, researchers should care about the known data quality issues and report the degree of quality (e.g., based on our measures framework) and possible threats in future publications.

Part VII

Glossary and Bibliography

Glossary

Acronym	Definition
#	Number of (used in tables)
API	Application Programming Interface
Basel II	Revised International Capital Framework
BTS	Bug Tracking System / Database
CMMI	Capability Maturity Model Integration
COBIT	Control Objectives for Information and Related Technology
CSS	Closed Source Software
CVS	Concurrent Versions System
DC	Data Characteristics
DQ	Data Quality
ESE	Empirical Software Engineering
IDE	Integrated Development Environment
IT	Information Technology
ITIL	Information Technology Infrastructure Library
KLOC	1 000 Lines of Code
LOC	Lines of Code
MR	Modification Request
OSS	Open Source Software
PR	Problem Report
RHDB	Release History Database
SE	Software Engineering
SOX	Sarbanes-Oxley Act
SPI	Software Process Improvement
SVN	Subversion
VCS	Version Control System
ZKB	Zurich Cantonal Bank

Bibliography

Agresti, A. and Coull, B. A. (1998). Approximate Is Better Than "Exact" for Interval Estimation of Binomial Proportions. *The American Statistician*, 52(2):119–126.

Ambroise, C. and McLachlan, G. J. (2002). Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the National Academy of Sciences*, 99(10):6562–6566.

Antoniol, G., Ayari, K., Penta, M. D., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a Bug or an Enhancement? A Text-based Approach to Classify Change Requests. In *CASCON '08: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research*, pages 304–318, Ontario, Canada. ACM.

Antoniol, G., Gall, H., Penta, M. D., and Pinzger, M. (2004). Mozilla: Closing the Circle. Technical Report TUV-1841-2004-05, Vienna University of Technology.

Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who Should Fix This Bug? In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 361–370, Shanghai, China. ACM.

APM Group Ltd (2010). Official ITIL Website. `http://www.itil-officialsite.com/`.

Aranda, J. and Venolia, G. (2009). The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories. In *ICSE '09: Proceeding of the 31st International Conference on Software Engineering*, pages 298–308, Vancouver, British Columbia, Canada. IEEE Computer Society.

Ashrafi, N. (2003). The impact of software process improvement on quality: in theory and practice. *Information Management*, 40(7):677–690.

Askari, M. and Holt, R. (2006). Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 126–132, Shanghai, China. ACM.

Aversano, L., Cerulo, L., and Grosso, C. D. (2007). Learning from Bug-introducing Changes to Prevent Fault Prone Code. In *IWPSE '07: Proceedings of the Ninth International Workshop on Principles of Software Evolution*, pages 19–26, Dubrovnik, Croatia. ACM.

Ayari, K., Meshkinfam, P., Antoniol, G., and DiPenta, M. (2007). Threats on Building Models from CVS and Bugzilla Repositories: the Mozilla Case Study. In *CASCON '07: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, pages 215–228, Ontario, Canada. ACM.

Bachmann, A. and Bernstein, A. (2009a). Data Retrieval, Processing and Linking for Software Process Data Analysis. Technical Report IFI-2009.0003, Dynamic and Distributed Information Systems Group, Department of Informatics, University of Zurich.

Bachmann, A. and Bernstein, A. (2009b). Software Process Data Quality and Characteristics - A Historical View on Open and Closed Source Projects. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 119–128, Amsterdam, The Netherlands. ACM.

Bachmann, A. and Bernstein, A. (2010). When Process Data Quality Affects the Number of Bugs: Correlations in Software Engineering Datasets. In *MSR '10: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 62–71, Cape Town, South Africa. IEEE Computer Society.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The Missing Links: Bugs and Bug-Fix Commits. In *ACM SIGSOFT / FSE '10: Proceedings of the eighteenth International Symposium on the Foundations of Software Engineering*, page to appear, Santa Fe, New Mexico, USA. ACM.

Bakota, T., Ferenc, R., Gyimothy, T., Riva, C., and Xu, J. (2006). Towards Portable Metrics-based Models for Software Maintenance Problems. In *ICSM '06: Proceedings of the International Conference on Software Maintenance*, pages 483–486, Philadelphia, Pennsylvania, USA. IEEE Computer Society.

Basel Committee on Banking Supervision (2006). *Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version*. Bank for International Settlements, Basel, Switzerland.

Basili, V. R., Bri, L., and Melo, W. L. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1:528–532.

Batini, C., Cappiello, C., Francalanci, C., and Maurino, A. (2009). Methodologies for Data Quality Assessment and Improvement. *ACM Computer Survey*, 41(3):1–52.

Benjamini, Y. and Hochberg, Y. (1995). Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300.

Berk, R. A. (1983). An Introduction to Sample Selection Bias in Sociological Data. *American Sociological Review*, 48(3):386–398.

Bernstein, A., Ekanayake, J., and Pinzger, M. (2007). Improving Defect Prediction Using Temporal Features and Non Linear Models. In *IWPSE '07: Proceedings of the Ninth International Workshop on Principles of Software Evolution*, pages 11–18, Dubrovnik, Croatia. ACM.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2007a). Quality of Bug Reports in Eclipse. In *eTX '07: In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, pages 21–25, Montreal, Quebec, Canada. ACM.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2007b). What Makes a Good Bug Report? Technical report, Saarland University, Saarbrücken, Germany.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008). Duplicate Bug Reports Considered Harmful... Really? In *ICSM '08: Proceedings of the International Conference on Software Maintenance*, pages 337–345, Beijing, China. IEEE Computer Society.

Binkley, A. B. and Schach, S. R. (1998). Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In *ICSE '98: In Proceedings of the 20th International Conference on Software Engineering*, pages 452–455, Kyoto, Japan. IEEE Computer Society.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009a). Fair and Balanced? Bias in Bug-Fix Datasets. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering*, pages 121–130, Amsterdam, The Netherlands. ACM.

Bird, C., Bachmann, A., Rahman, F., and Bernstein, A. (2010). LINKSTER: Enabling Efficient Manual Inspection and Annotation of Mined Data. In *ACM SIGSOFT / FSE '10: Proceedings of the eighteenth International Symposium on the Foundations of Software Engineer-*

ing, page to appear, Santa Fe, New Mexico, USA. ACM. formal demonstration.

Bird, C., Nagappan, N., Devanbu, P., Gall, H., and Murphy, B. (2009b). Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 518–528, Vancouver, British Columbia, Canada. IEEE Computer Society.

Breu, S. and Zimmermann, T. (2006). Mining Aspects from Version History. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, Tokyo, Japan. IEEE Computer Society.

Breu, S., Zimmermann, T., and Lindig, C. (2006). Mining Eclipse for Cross-Cutting Concerns. In *MSR '06: Proceedings of the 2006 international Workshop on Mining software repositories*, pages 94–97, Shanghai, China. ACM.

Canfora, G. and Cerulo, L. (2005). Impact analysis by mining software and change request repositories. In *METRICS '05: Proceedings of the 11th International Symposium on Software Metrics*, pages 29–38, Como, Italy. IEEE Computer Society.

Cartwright, M. H., Shepperd, M. J., and Song, Q. (2003). Dealing with Missing Software Project Data. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 154, Sydney, Australia. IEEE Computer Society.

Catal, C. and Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346–7354.

Cavano, J. P. and McCall, J. A. (1978). A framework for the measurement of software quality. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 133–139. ACM.

Chen, K., Schach, S. R., Yu, L., Offutt, J., and Heller, G. Z. (2004). Open-Source Change Logs. *Empirical Software Engineering*, 9(3):197–210.

CMMI Product Team (2006). *CMMI® for Development, Version 1.2*. Carnegie Mellon, Software Engineering Institute, Pittsburgh, PA, USA.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum Associates, Inc., New Jersey, 2nd edition.

Conover, W. J. (1998). *Practical Nonparametric Statistics*. John Wiley & Sons, 3rd edition.

Crowston, K. (1997). A Coordination Theory Approach to Organizational Process Design. *Organization Science*, 8(2):157–175.

Dallmeier, V. and Zimmermann, T. (2007). Extraction of Bug Localization Benchmarks from History. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 433–436, Atlanta, Georgia, USA. ACM.

D'Ambros, M., Lanza, M., and Gall, H. C. (2005). Fractal Figures: Visualizing Development Effort for CVS Entities. In *VISSOFT '05: Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis*, pages 46–51, Budapest, Hungary. IEEE CS Press.

Diaz, M. and Sligo, J. (1997). How Software Process Improvement Helped Motorola. *IEEE Software*, 14(5):75–81.

Dowdy, S., Wearden, S., and Chilko, D. (2004). *Statistics for Research*. Wiley-Interscience, 3rd edition.

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515.

Easterbrook, P. J., Berlin, J. A., Gopalan, R., and Matthews, D. R. (1991). Publication bias in clinical research. *The Lancet*, 337(8746):867–872.

Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2007). Selecting Empirical Methods for Software Engineering Research. *Guide to Advanced Empirical Software Engineering*, Section III:285–311.

Ekanayake, J., Tappolet, J., Gall, H. C., and Bernstein, A. (2009). Tracking Concept Drift of Software Projects Using Defect Prediction Quality. In *MSR '09: Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, pages 51–60, Vancouver, Canada. IEEE Computer Society.

Federal Office for Information Security (BSI) (2005). *IT-Grundschutz Catalogues*. Germany.

Fischer, M. and Gall, H. C. (2006). EvoGraph: A Lightweight Approach to Evolutionary and Structural Analysis of Large Software Systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 179–188, Benevento, Italy. IEEE Computer Society.

Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and Relating Bug Report Data for Feature Tracking. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, pages 90–99, Victoria, British Columbia, Canada. IEEE Computer Society.

Fischer, M., Pinzger, M., and Gall, H. C. (2003b). Populating a Release History Database from Version Control and Bug Tracking Systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands. IEEE Computer Society.

Gasser, L. and Ripoche, G. (2003). Distributed Collective Practices and Free/Open-Source Software Problem Management: Perspectives and Methods. In *CITE '03: Proceedings of the Conference on Coopération, Innovations et Technologies*, pages 349–365.

Gegick, M., Rotella, P., and Xie, T. (2010). Identifying Security Bug Reports via Text Mining: An Industrial Case Study. In *MSR '10: Proceedings of the 7th IEEE Working Conference on Mining Software*

Repositories, pages 11–20, Cape Town, South Africa. IEEE Computer Society.

German, D. M. (2004). Mining CVS repositories, the softChange experience. In *MSR '04: Proceeding of the 1st International Workshop on Mining Software Repositories*, pages 17–21, Edinburgh, Scotland, UK. ACM.

Godfrey, M. W., Hassan, A. E., Herbsleb, J., Murphy, G. C., Robillard, M., Devanbu, P., Mockus, A., Perry, D. E., and Notkin, D. (2009). Future of Mining Software Archives: A Roundtable. *IEEE Software*, 26(1):67–70.

Grabe, M. E., Zhou, S., and Barnett, B. (2001). Explicating Sensationalism in Television News: Content and the Bells and Whistles of Form. *Journal of Broadcasting & Electronic Media*, 45(4):635–655.

Grady, R. B. and Caswell, D. L. (1987). *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall PTR.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7):653–661.

Gyimothy, T., Ferenc, R., and Siket, I. (2005). Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910.

Harter, D. E. and Slaughter, S. A. (2000). Process maturity and software quality: a field study. In *ICIS '00: Proceedings of the twenty first international conference on Information systems*, pages 407–411, Brisbane, Queensland, Australia. Association for Information Systems.

Hassan, A. E. and Holt, R. C. (2005). The Top Ten List: Dynamic Fault Prediction. In *ICSM '05: Proceedings of the International Conference on Software Maintenance*, pages 263–272, Budapest, Hungary. IEEE Computer Society.

Heckman, J. J. (1979). Sample Selection Bias as a Specification Error. *Econometrica*, 47(1):153–161.

Herraiz, I., Robles, G., and Gonzalez-Barahona, J. M. (2005). Towards Predictor Models for large Libre Software Projects. In *PROMISE '05: Proceedings of the 2005 workshop on Predictor models in software engineering*, pages 1–6, St. Louis, Missouri, USA. ACM.

Hooimeijer, P. and Weimer, W. (2007). Modeling Bug Report Quality. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 34–43, Atlanta, Georgia, USA. ACM.

ISO/IEC (2005a). ISO/IEC 27002: Information technology - Security techniques - Code of practice for information security management.

ISO/IEC (2005b). ISO/IEC 9000:2005: Quality management systems – Fundamentals and vocabulary.

IT Governance Institute (2007). *Cobit 4.1*. Isaca.

Joshi, H., Zhang, C., Ramaswamy, S., and Bayrak, C. (2007). Local and Global Recency Weighting Approach to Bug Prediction. In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 33–34, Minneapolis, Minnesota, USA. IEEE Computer Society.

Just, S., Premraj, R., and Zimmermann, T. (2008). Towards the Next Generation of Bug Tracking Systems. In *VL/HCC '08: Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 82–85, Herrsching am Ammersee, Germany. IEEE Computer Society.

Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering*. Addison Wesley, 2nd edition.

Kendall, M. G. (1938). A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93.

Kim, S., Pan, K., and Whitehead, E. J. (2006a). Memories of Bug Fixes. In *FSE '06: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, Portland, Oregon, USA. ACM.

Kim, S. and Whitehead, E. J. (2006). How Long Did It Take To Fix Bugs? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 173–174, Shanghai, China. ACM.

Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. (2006b). Automatic Identification of Bug-Introducing Changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Tokyo, Japan. IEEE Computer Society.

Kim, S., Zimmermann, T., Whitehead, E. J., and Zeller, A. (2007). Predicting Faults from Cached History. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Minneapolis, Minnesota, USA. IEEE Computer Society.

Klein, M. (1993). Capturing Design Rationale in Concurrent Engineering Teams. *IEEE Computer*, 26(1):39–47.

Knab, P., Pinzger, M., and Bernstein, A. (2006). Predicting Defect Densities in Source Code Files with Decision Tree Learners. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 119–125, Shanghai, China. ACM.

Ko, A. J., Myers, B. A., and Chau, D. H. (2006). A Linguistic Analysis of How People Describe Software Problems. In *VL/HCC '08: Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 127–134, Herrsching am Ammersee, Germany. IEEE Computer Society.

Koru, A. G. and Tian, J. (2004). Defect Handling in Medium and Large Open Source Projects. *IEEE Software*, 21(4):54–61.

Koru, A. G. and Tian, J. (2005). Comparing high-change modules and modules with the highest measurement values in two large-scale

open-source products. *IEEE Transactions on Software Engineering*, 31(8):625–642.

Kroeger, T. and Davidson, N. (2009). A Perspective-Based Model of Quality for Software Engineering Processes. In *ASWEC '09: Proceedings of the 2009 Australian Software Engineering Conference*, pages 152–161, Gold Coast, Australia. IEEE Computer Society.

Lamkanfi, A., Demeyer, S., Giger, E., and Goethals, B. (2010). Predicting the Severity of a Reported Bug. In *MSR '10: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 1–10, Cape Town, South Africa. IEEE Computer Society.

Lanza, M. (2003). *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, Switzerland.

Levy, M. R. (1983). The Methodology and Performance of Election Day Polls. *The Public Opinion Quarterly*, 47(1):54–67.

Liebchen, G., Twala, B., Shepperd, M., Cartwright, M., and Stephens, M. (2007). Filtering, Robust Filtering, Polishing: Techniques for Addressing Quality in Software Data. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 99–106, Madrid, Spain. IEEE Computer Society.

Liebchen, G. A. and Shepperd, M. (2008). Data Sets and Data Quality in Software Engineering. In *PROMISE '08: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, pages 39–44, Leipzig, Germany. ACM.

Linstead, E. and Baldi, P. (2009). Mining the coherence of GNOME bug reports with statistical topic models. In *MSR '09: Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, pages 99–102, Vancouver, Canada. IEEE Computer Society.

MacKinnon, J. G. and Smith, A. A. (August 1998). Approximate Bias Correction in Econometrics. *Journal of Econometrics*, 85(2):205–230.

Michaud, J., Storey, M.-A., and Muller, H. (2001). Integrating information sources for visualizing Java programs. In *ICSM '01: Proceedings of the International Conference on Software Maintenance*, pages 250–258, Florence, Italy. IEEE Computer Society.

Mitchell, T. M. (1997). *Machine Learning*. McGraw Hill Higher Education.

Mockus, A. (2008). Missing Data in Software Engineering. *Guide to Advanced Empirical Software Engineering, Springer London*, Section II:185–200.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions of Software Engineering Methodologies*, 11(3):309–346.

Mockus, A. and Votta, L. G. (2000). Identifying Reasons for Software Changes Using Historic Databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, pages 120–130, San Jose, California, USA. IEEE Computer Society.

Moser, R., Pedrycz, W., and Succi, G. (2008). A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *ICSE '08: Proceedings of the 30th international conference on Software Engineering*, pages 181–190, Leipzig, Germany. ACM.

Mozilla Foundation, Bugzilla (2010). The Bugzilla Guide: Life Cycle of a Bug. `http://www.bugzilla.org/docs/3.6/en/html/lifecycle.html`.

Nagappan, N. and Ball, T. (2005). Use of Relative Code Churn Measures to Predict System Defect Density. In *ICSE '05: Proceedings of the International Conference on Software Engineering*, pages 284–292, St. Louis, Missouri, USA. ACM.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining Metrics to Predict Component Failures. In *ICSE '06: Proceedings of the 28th International*

Conference on Software Engineering, pages 452–461, Shanghai, China. ACM.

Netcraft Ltd. (2010). July 2010 Web Server Survey. `http://news.netcraft.com/archives/2010/07/16/july-2010-web-server-survey-16.html`.

Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. (2007). Predicting Vulnerable Software Components. In *CCS '07: Proceedings of the ACM conference on Computer and communications security*.

Nick, M. and Tautz, C. (1999). Practical Evaluation of an Organizational Memory Using the Goal-Question-Metric Technique. *Lecture notes in computer science*, 1570:138–147.

Nickerson, R. S. (1998). Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. *Review of General Psychology*, 2(2):175–220.

Ohlsson, N. and Alberg, H. (1996). Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions on Software Engineering*, 22(12):886–894.

Ostrand, M.-T. J., Weyuker, F.-E. J., and Bell, R. M. (2005). Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31(4):340–355.

Panjer, L. D. (2007). Predicting Eclipse Bug Lifetimes. In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, page 29, Minneapolis, Minnesota, USA. IEEE Computer Society.

Paulson, J. W., Succi, G., and Eberlein, A. (2004). An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering*, 30(4):246–256.

Pinzger, M. (2005). *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, University of Vienna, Austria.

Pinzger, M., Gall, H., Fischer, M., and Lanza, M. (2005). Visualizing Multiple Evolution Metrics. In *Softvis '05: Proceedings of the ACM*

Symposium on Software Visualization, pages 67–75, St. Louis, Missouri, USA. ACM.

Rahman, F., Bird, C., and Devanbu, P. (2010). Clones: What *is* that Smell? In *MSR '10: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 72–81, Cape Town, South Africa. IEEE Computer Society.

Ratzinger, J., Fischer, M., and Gall, H. C. (2005). EvoLens: Lens-View Visualizations of Evolution Data. In *IWPSE '05: Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal. IEEE Computer Society Press.

Ratzinger, J., Sigmund, T., Vorburger, P., and Gall, H. C. (2007). Mining Software Evolution to Predict Refactoring. In *ESEM '07: Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 354–363, Madrid, Spain. IEEE Computer Society.

Royce, W. W. (1970). Managing the Development of Large Software Systems. *IEEE Westcon*, pages 1–9. Reprinted in Proceedings of the Ninth International Conference on Software Engineering (ICSE), March 1987, pp. 328–338.

Sahoo, S. K., Criswell, J., and Adve, V. (2010). Towards Automated Bug Diagnosis: An Empirical Study of Reported Software Bugs in Server Applications. In *ICSE '10: Proceedings of the 2010 IEEE 32nd International Conference on Software Engineering*, pages 485–494, Cape Town, South Africa. to appear.

Schackmann, H. and Lichter, H. (2009). Evaluating process quality in GNOME based on change request data. In *MSR '09: Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, pages 95–98, Vancouver, Canada. IEEE Computer Society.

Schröter, A., Zimmermann, T., Premraj, R., and Zeller, A. (2006a). If Your Bug Database Could Talk... In *ISESE '06: Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, pages 18–20, Rio de Janeiro, Brazil. ACM.

Schröter, A., Zimmermann, T., and Zeller, A. (2006b). Predicting Component Failures at Design Time. In *ISESE '06: Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–27, Rio de Janeiro, Brazil. ACM.

Schugerl, P., Rilling, J., and Charland, P. (2008). Mining Bug Repositories – A Quality Assessment. In *CIMCA '08: Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control & Automation*, pages 1105–1110, Vienna, Austria. IEEE Computer Society.

Singleton, R. A. and Straits, B. C. (2009). *Approaches to Social Research*. Oxford University Press, USA, 5th edition.

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When Do Changes Induce Fixes? In *MSR '05: Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 24–28, Saint Louis, Missouri, USA. ACM.

Tajima, D. and Matsubara, T. (1981). Special Feature – The Computer Software Industry in Japan. *Computer*, 14(5):89–96.

Terwilliger, J. D. and Weiss, K. M. (2003). Confounding, ascertainment bias, and the blind quest for a genetic 'fountain of youth'. *Annals of Medicine*, 35(7):532–544.

United States Code (2002). Sarbanes-Oxley Act of 2002, PL 107-204, 116 Stat 745. Codified in Sections 11, 15, 18, 28, and 29 USC.

Čubranić, D. and Murphy, G. C. (2003). Hipikat: Recommending Pertinent Software Development Artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Portland, Oregon, USA. IEEE Computer Society.

Čubranić, D., Murphy, G. C., Singer, J., and Booth, K. S. (2005). Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465.

Weiss, C., Premraj, R., Zimmermann, T., and Zeller, A. (2007). How Long will it Take to Fix This Bug? In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, page 1, Minneapolis, Minnesota, USA. IEEE Computer Society.

Williams, C. C. and Hollingsworth, J. K. (2004). Bug Driven Bug Finders. In *MSR '04: Proceeding of the 1st International Workshop on Mining Software Repositories*, pages 70–74, Edinburgh, Scotland, UK. ACM.

Yu, L. and Chen, K. (2007). Evaluating the Post-Delivery Fault Reporting and Correction Process in Closed-Source and Open-Source Software. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops, Fifth International Workshop on Software Quality (WoSQ'07)*, pages 108–113, Minneapolis, Minnesota, USA. IEEE Computer Society.

Zadrozny, B. (2004). Learning and Evaluating Classifiers under Sample Selection Bias. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, pages 114–122, Banff, Alberta, Canada. ACM.

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting Defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 1–9, Minneapolis, Minnesota, USA. IEEE Computer Society.

Zimmermann, T. and Weissgerber, P. (2004). Preprocessing CVS Data for Fine-Grained Analysis. In *MSR '04: Proceeding of the 1st International Workshop on Mining Software Repositories*, pages 2–6, Edinburgh, Scotland, UK. ACM.

Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2004). Mining Version Histories to Guide Software Changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Edinburgh, Scotland, UK. IEEE Computer Society.

Zuboff, S. (1988). *In the Age of the Smart Machine: The Future of Work and Power*. Basic Books, 3rd edition.

Part VIII

Appendix

A

Curriculum Vitae

Personal Details

Name:	Adrian Joe Ernst Bachmann
Date of Birth:	July 30, 1981
Place of Birth:	Lucerne, Switzerland
Citizenship:	Swiss

Education

08/2006 – 08/2010	Doctoral student at University of Zurich, Department of Informatics. Graduated with a Doctorate of Informatics (Dr. Inform.).
10/2002 – 03/2006	Computer Science at University of Zurich. Graduated with a Diploma in Informatics (Dipl. Inform.).
08/1993 – 07/2001	Swiss high school in Sursee, Switzerland.
08/1987 – 07/1993	Swiss elementary school in Eich, Switzerland.

Professional Experience

12/2009 – present	Analyst Operational Risk at Zürcher Kantonalbank.
04/2006 – 11/2009	Project Leader at Zürcher Kantonalbank.
07/2005 – 11/2005	Internship at Zürcher Kantonalbank.
09/1998 – 12/2005	Chief Information Officer at Marc Daniel Personalberatung.

Miscellaneous

08/2001 – 10/2002	Swiss Military Service